UNIX USER'S MANUAL

Supplementary Documents

~~Samuel Rebelsky~~
~~947-0613~~

Eric Fischer

Cover design by John Lassetter, Lucasfilm, Ltd.

First Printing	July 1984
Second Printing	December 1984

UNIX USER'S MANUAL
Supplementary Documents

4.2 Berkeley Software Distribution
Virtual VAX−11 Version

March, 1984

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

This manual reflects system enhancements made at Berkeley and sponsored in part by NSF Grants MCS-7807291, MCS-8005144, and MCS-74-07644-A04; DOE Contract DE-AT03-76SF00034 and Project Agreement DE-AS03-79ER10358; and by Defense Advanced Research Projects Agency (DoD) ARPA Order No. 4031, Monitored by Naval Electronics Systems Command under Contract No. N00039-80-K-0649.

PREFACE

This manual is part of a five volume set intended for use with the 4.2 Berkeley Software Distribution for the VAX-11 computer. While the five volumes together contain virtually the same material presented in the four volume UNIX Programmer's Manual distributed with 4.2BSD, the manuals reflect a revised organization necessitated by the large quantity of information. The documentation is divided into three logically distinct *manuals*:

- UNIX User's Manual,
- UNIX Programmer's Manual, and
- UNIX System Manager's Manual.

Each of the User and Programmer manuals are two volumes: a Reference Guide, containing relevant sections from Volume 1 of the old UNIX Programmer's Manual, and a volume of Supplementary Documents, containing pertinent material from Volume 2 of the old UNIX Programmer's Manual. The System Manager's manual consists of a single volume containing information from both Volumes 1 and 2. We acknowledge those who have assisted us in putting together these manuals. In particular, we thank Tom Ferrin for pursuing the printing particulars.

M. J. Karels
S. J. Leffler

Preface to the 4.2 Berkeley distribution

This update to the 4.1 distribution of June 1981 provides support for the VAX 11/730, full networking and interprocess communication support, an entirely new file system, and many other new features. It is certainly the most ambitious release of software ever prepared here and represents many man-years of work. Bill Shannon (both at DEC and at Sun Microsystems) and Robert Elz of the University of Melbourne contributed greatly to this distribution through new device drivers and painful debugging episodes. Rob Gurwitz of BBN wrote the initial version of the code upon which the current networking support is based. Eric Allman of Britton-Lee donated countless hours to the mail system. Bill Croft (both at SRI and Sun Microsystems) aided in the debugging and development of the networking facilities. Dennis Ritchie of Bell Laboratories also contributed greatly to this distribution, providing valuable advise and guidance. Helge Skrivervik worked on the device drivers which enabled the distribution to be delivered with a TU58 console cassette and RX01 console flopppy disk, and rewrote major portions of the standalone i/o system to support formatting of non-DEC peripherals.

Numerous others contributed their time and energy in organizing the user software for release, while many groups of people on campus suffered patiently through the low spots of development. As always, we are grateful to the UNIX user community for encouragement and support.

Once again, the financial support of the Defense Advanced Research Projects Agency is gratefully acknowledged.

S. J. Leffler
W. N. Joy
M. K. McKusick

UNIX User's Manual

Supplementary Documents

4.2 Berkeley Software Distribution, Virtual VAX−11 Version

March, 1984

This volume contains documents which supplement the information in The UNIX† User's Manual Reference Guide for the Virtual VAX-11 version of the system as distributed by U.C. Berkeley.

† UNIX is a trademark of Bell Laboratories.

12. Edit: A Tutorial (Revised)
    For those who prefer line oriented editing, an introduction assuming no previous knowledge of UNIX or of text editing.
13. An Introduction to Display Editing with Vi.
    The document to read to learn to use the *vi* screen editor.
14. Ex Reference Manual (Version 3. — Oct. 1980).
    The final reference for the *ex* editor, which underlies both *edit* and *vi*. Also includes "Ex Changes — Version 3.1 to 3.5".
15. SED — A Non-interactive Text Editor.
    A variant of the editor for processing large inputs.
16. AWK — A Pattern Scanning and Processing Language.
    Makes it easy to specify many data transformations and selection operations.

Document Preparation

17. Typesetting Documents on the UNIX System.
    Describes the basic use of the formatting tools. Also describes "—ms"; a standardized package of formatting requests that can be used to lay out most documents (including this volume).
18. A Revised Version of —ms.
    A quick description of the revisions made to the —ms formatting macros for *nroff* and *troff*.
19. Writing Papers with NROFF using —me.
    A popular macro package for *nroff*.
20. —me Reference Manual
    The final word on —me.
21. A System for Tyepsetting Mathematics.
    Describes EQN an easy-to-learn language for doing high-quality mathematical typesetting.
22. Tyepsetting Mathematics — User's Guide (Second Edition).
    This is the user's guide for EQN.
23. TBL — A Program to Format Tables.
    A program to permit easy specification of tabular material for typesetting. Again, easy to learn and use.
24. Some Applications of Inverted Indexes on the UNIX System.
    Describes, among other things, the program *refer* which fills in bibliographic citations from a data base automatically.
25. Refer — A Bibliography System.
    An introduction to the tools used to maintain bibliographic databases. The major program, *refer*, is used to automatically retrieve and format references based on document citations.
26. Writing Tools — the Style and Diction Programs.
    Description of programs which help you understand and improve your writing style.
27. NROFF/TROFF User's Manual.
    The basic formatting program.
28. A TROFF Tutorial.
    An introduction to *troff* for those who really want to know such things.
29. The Berkeley Font Catalog
    Samples of fonts currently available for the raster plotters.

7th Edition UNIX — Summary

September 6, 1978

Bell Laboratories
Murray Hill, New Jersey 07974

A. What's new: highlights of the 7th edition UNIX† System

Aimed at larger systems. Devices are addressable to 2^{31} bytes, files to 2^{30} bytes. 128K memory (separate instruction and data space) is needed for some utilities.

Portability. Code of the operating system and most utilities has been extensively revised to minimize its dependence on particular hardware.

Fortran 77. F77 compiler for the new standard language is compatible with C at the object level. A Fortran structurer, STRUCT, converts old, ugly Fortran into RATFOR, a structured dialect usable with F77.

Shell. Completely new SH program supports string variables, trap handling, structured programming, user profiles, settable search path, multilevel file name generation, etc.

Document preparation. TROFF phototypesetter utility is standard. NROFF (for terminals) is now highly compatible with TROFF. MS macro package provides canned commands for many common formatting and layout situations. TBL provides an easy to learn language for preparing complicated tabular material. REFER fills in bibliographic citations from a data base.

UNIX-to-UNIX file copy. UUCP performs spooled file transfers between any two machines.

Data processing. SED stream editor does multiple editing functions in parallel on a data stream of indefinite length. AWK report generator does free-field pattern selection and arithmetic operations.

Program development. MAKE controls re-creation of complicated software, arranging for minimal recompilation.

Debugging. ADB does postmortem and breakpoint debugging, handles separate instruction and data spaces, floating point, etc.

C language. The language now supports definable data types, generalized initialization, block structure, long integers, unions, explicit type conversions. The LINT verifier does strong type checking and detection of probable errors and portability problems even across separately compiled functions.

Lexical analyzer generator. LEX converts specification of regular expressions and semantic actions into a recognizing subroutine. Analogous to YACC.

Graphics. Simple graph-drawing utility, graphic subroutines, and generalized plotting filters adapted to various devices are now standard.

Standard input-output package. Highly efficient buffered stream I/O is integrated with formatted input and output.

Other. The operating system and utilities have been enhanced and freed of restrictions in many other ways too numerous to relate.

† UNIX is a Trademark of Bell Laboratories.

B. Hardware

The 7th edition UNIX operating system runs on a DEC PDP-11/45 or 11/70* with at least the following equipment:

128K to 2M words of managed memory; parity not used.

disk: RP03, RP04, RP06, RK05 (more than 1 RK05) or equivalent.

console typewriter.

clock: KW11-L or KW11-P.

The following equipment is strongly recommended:

communications controller such as DL11 or DH11.

full duplex 96-character ASCII terminals.

9-track tape or extra disk for system backup.

The system is normally distributed on 9-track tape. The minimum memory and disk space specified is enough to run and maintain UNIX. More will be needed to keep all source on line, or to handle a large number of users, big data bases, diversified complements of devices, or large programs. The resident code occupies 12-20K words depending on configuration; system data occupies 10-28K words.

There is no commitment to provide 7th edition UNIX on PDP-11/34, 11/40 and 11/60 hardware.

C. Software

Most of the programs available as UNIX commands are listed. Source code and printed manuals are distributed for all of the listed software except games. Almost all of the code is written in C. Commands are self-contained and do not require extra setup information, unless specifically noted as "interactive." Interactive programs can be made to run from a prepared script simply by redirecting input. Most programs intended for interactive use (e.g., the editor) allow for an escape to command level (the Shell). Most file processing commands can also go from standard input to standard output ("filters"). The piping facility of the Shell may be used to connect such filters directly to the input or output of other programs.

1. Basic Software

This includes the time-sharing operating system with utilities, a machine language assembler and a compiler for the programming language C—enough software to write and run new applications and to maintain or modify UNIX itself.

1.1. Operating System

□ UNIX The basic resident code on which everything else depends. Supports the system calls, and maintains the file system. A general description of UNIX design philosophy and system facilities appeared in the Communications of the ACM, July, 1974. A more extensive survey is in the Bell System Technical Journal for July-August 1978. Capabilities include:

- O Reentrant code for user processes.
- O Separate instruction and data spaces.
- O "Group" access permissions for cooperative projects, with overlapping memberships.
- O Alarm-clock timeouts.

*PDP is a Trademark of Digital Equipment Corporation.

○ Timer-interrupt sampling and interprocess monitoring for debugging and measurement.
○ Multiplexed I/O for machine-to-machine communication.

□ DEVICES All I/O is logically synchronous. I/O devices are simply files in the file system. Normally, invisible buffering makes all physical record structure and device characteristics transparent and exploits the hardware's ability to do overlapped I/O. Unbuffered physical record I/O is available for unusual applications. Drivers for these devices are available; others can be easily written:
○ Asynchronous interfaces: DH11, DL11. Support for most common ASCII terminals.
○ Synchronous interface: DP11.
○ Automatic calling unit interface: DN11.
○ Line printer: LP11.
○ Magnetic tape: TU10 and TU16.
○ DECtape: TC11.
○ Fixed head disk: RS11, RS03 and RS04.
○ Pack type disk: RP03, RP04, RP06; minimum-latency seek scheduling.
○ Cartridge-type disk: RK05, one or more physical devices per logical device.
○ Null device.
○ Physical memory of PDP-11, or mapped memory in resident system.
○ Phototypesetter: Graphic Systems System/1 through DR11C.

□ BOOT Procedures to get UNIX started.

□ MKCONF Tailor device-dependent system code to hardware configuration. As distributed, UNIX can be brought up directly on any acceptable CPU with any acceptable disk, any sufficient amount of core, and either clock. Other changes, such as optimal assignment of directories to devices, inclusion of floating point simulator, or installation of device names in file system, can then be made at leisure.

1.2. User Access Control

□ LOGIN Sign on as a new user.
○ Verify password and establish user's individual and group (project) identity.
○ Adapt to characteristics of terminal.
○ Establish working directory.
○ Announce presence of mail (from MAIL).
○ Publish message of the day.
○ Execute user-specified profile.
○ Start command interpreter or other initial program.

□ PASSWD Change a password.
○ User can change his own password.
○ Passwords are kept encrypted for security.

□ NEWGRP Change working group (project). Protects against unauthorized changes to projects.

1.3. Terminal Handling

□ TABS Set tab stops appropriately for specified terminal type.

□ STTY Set up options for optimal control of a terminal. In so far as they are deducible from the input, these options are set automatically by LOGIN.

- Half vs. full duplex.
- Carriage return + line feed vs. newline.
- Interpretation of tabs.
- Parity.
- Mapping of upper case to lower.
- Raw vs. edited input.
- Delays for tabs, newlines and carriage returns.

1.4. File Manipulation

☐ CAT	Concatenate one or more files onto standard output. Particularly used for unadorned printing, for inserting data into a pipeline, and for buffering output that comes in dribs and drabs. Works on any file regardless of contents.
☐ CP	Copy one file to another, or a set of files to a directory. Works on any file regardless of contents.
☐ PR	Print files with title, date, and page number on every page. ○ Multicolumn output. ○ Parallel column merge of several files.
☐ LPR	Off-line print. Spools arbitrary files to the line printer.
☐ CMP	Compare two files and report if different.
☐ TAIL	Print last *n* lines of input ○ May print last *n* characters, or from *n* lines or characters to end.
☐ SPLIT	Split a large file into more manageable pieces. Occasionally necessary for editing (ED).
☐ DD	Physical file format translator, for exchanging data with foreign systems, especially IBM 370's.
☐ SUM	Sum the words of a file.

1.5. Manipulation of Directories and File Names

☐ RM	Remove a file. Only the name goes away if any other names are linked to the file. ○ Step through a directory deleting files interactively. ○ Delete entire directory hierarchies.
☐ LN	"Link" another name (alias) to an existing file.
☐ MV	Move a file or files. Used for renaming files.
☐ CHMOD	Change permissions on one or more files. Executable by files' owner.
☐ CHOWN	Change owner of one or more files.
☐ CHGRP	Change group (project) to which a file belongs.
☐ MKDIR	Make a new directory.
☐ RMDIR	Remove a directory.
☐ CD	Change working directory.
☐ FIND	Prowl the directory hierarchy finding every file that meets specified criteria.

○ Criteria include:
  name matches a given pattern,
  creation date in given range,
  date of last use in given range,
  given permissions,
  given owner,
  given special file characteristics,
  boolean combinations of above.
○ Any directory may be considered to be the root.
○ Perform specified command on each file found.

1.6. Running of Programs

□ SH — The Shell, or command language interpreter.
○ Supply arguments to and run any executable program.
○ Redirect standard input, standard output, and standard error files.
○ Pipes: simultaneous execution with output of one process connected to the input of another.
○ Compose compound commands using:
  if ... then ... else,
  case switches,
  while loops,
  for loops over lists,
  break, continue and exit,
  parentheses for grouping.
○ Initiate background processes.
○ Perform Shell programs, i.e., command scripts with substitutable arguments.
○ Construct argument lists from all file names satisfying specified patterns.
○ Take special action on traps and interrupts.
○ User-settable search path for finding commands.
○ Executes user-settable profile upon login.
○ Optionally announces presence of mail as it arrives.
○ Provides variables and parameters with default setting.

□ TEST — Tests for use in Shell conditionals.
○ String comparison.
○ File nature and accessibility.
○ Boolean combinations of the above.

□ EXPR — String computations for calculating command arguments.
○ Integer arithmetic
○ Pattern matching

□ WAIT — Wait for termination of asynchronously running processes.

□ READ — Read a line from terminal, for interactive Shell procedure.

□ ECHO — Print remainder of command line. Useful for diagnostics or prompts in Shell programs, or for inserting data into a pipeline.

□ SLEEP — Suspend execution for a specified time.

□ NOHUP — Run a command immune to hanging up the terminal.

□ NICE — Run a command in low (or high) priority.

□ KILL	Terminate named processes.
□ CRON	Schedule regular actions at specified times. ○ Actions are arbitrary programs. ○ Times are conjunctions of month, day of month, day of week, hour and minute. Ranges are specifiable for each.
□ AT	Schedule a one-shot action for an arbitrary time.
□ TEE	Pass data between processes and divert a copy into one or more files.

1.7. Status Inquiries

□ LS	List the names of one, several, or all files in one or more directories. ○ Alphabetic or temporal sorting, up or down. ○ Optional information: size, owner, group, date last modified, date last accessed, permissions, i-node number.
□ FILE	Try to determine what kind of information is in a file by consulting the file system index and by reading the file itself.
□ DATE	Print today's date and time. Has considerable knowledge of calendric and horological peculiarities. ○ May set UNIX's idea of date and time.
□ DF	Report amount of free space on file system devices.
□ DU	Print a summary of total space occupied by all files in a hierarchy.
□ QUOT	Print summary of file space usage by user id.
□ WHO	Tell who's on the system. ○ List of presently logged in users, ports and times on. ○ Optional history of all logins and logouts.
□ PS	Report on active processes. ○ List your own or everybody's processes. ○ Tell what commands are being executed. ○ Optional status information: state and scheduling info, priority, attached terminal, what it's waiting for, size.
□ IOSTAT	Print statistics about system I/O activity.
□ TTY	Print name of your terminal.
□ PWD	Print name of your working directory.

1.8. Backup and Maintenance

□ MOUNT	Attach a device containing a file system to the tree of directories. Protects against nonsense arrangements.
□ UMOUNT	Remove the file system contained on a device from the tree of directories. Protects against removing a busy device.
□ MKFS	Make a new file system on a device.
□ MKNOD	Make an i-node (file system entry) for a special file. Special files are physical devices, virtual devices, physical memory, etc.

□ TP	
□ TAR	Manage file archives on magnetic tape or DECtape. TAR is newer. ○ Collect files into an archive. ○ Update DECtape archive by date. ○ Replace or delete DECtape files. ○ Print table of contents. ○ Retrieve from archive.
□ DUMP	Dump the file system stored on a specified device, selectively by date, or indiscriminately.
□ RESTOR	Restore a dumped file system, or selectively retrieve parts thereof.
□ SU	Temporarily become the super user with all the rights and privileges thereof. Requires a password.
□ DCHECK	
□ ICHECK	
□ NCHECK	Check consistency of file system. ○ Print gross statistics: number of files, number of directories, number of special files, space used, space free. ○ Report duplicate use of space. ○ Retrieve lost space. ○ Report inaccessible files. ○ Check consistency of directories. ○ List names of all files.
□ CLRI	Peremptorily expunge a file and its space from a file system. Used to repair damaged file systems.
□ SYNC	Force all outstanding I/O on the system to completion. Used to shut down gracefully.

1.9. Accounting

The timing information on which the reports are based can be manually cleared or shut off completely.

□ AC	Publish cumulative connect time report. ○ Connect time by user or by day. ○ For all users or for selected users.
□ SA	Publish Shell accounting report. Gives usage information on each command executed. ○ Number of times used. ○ Total system time, user time and elapsed time. ○ Optional averages and percentages. ○ Sorting on various fields.

1.10. Communication

□ MAIL	Mail a message to one or more users. Also used to read and dispose of incoming mail. The presence of mail is announced by LOGIN and optionally by SH. ○ Each message can be disposed of individually. ○ Messages can be saved in files or forwarded.

□ CALENDAR Automatic reminder service for events of today and tomorrow.

□ WRITE Establish direct terminal communication with another user.

□ WALL Write to all users.

□ MESG Inhibit receipt of messages from WRITE and WALL.

□ CU Call up another time-sharing system.
- ○ Transparent interface to remote machine.
- ○ File transmission.
- ○ Take remote input from local file or put remote output into local file.
- ○ Remote system need not be UNIX.

□ UUCP UNIX to UNIX copy.
- ○ Automatic queuing until line becomes available and remote machine is up.
- ○ Copy between two remote machines.
- ○ Differences, mail, etc., between two machines.

1.11. Basic Program Development Tools

Some of these utilities are used as integral parts of the higher level languages described in section 2.

□ AR Maintain archives and libraries. Combines several files into one for housekeeping efficiency.
- ○ Create new archive.
- ○ Update archive by date.
- ○ Replace or delete files.
- ○ Print table of contents.
- ○ Retrieve from archive.

□ AS Assembler. Similar to PAL-11, but different in detail.
- ○ Creates object program consisting of
  - code, possibly read-only,
  - initialized data or read-write code,
  - uninitialized data.
- ○ Relocatable object code is directly executable without further transformation.
- ○ Object code normally includes a symbol table.
- ○ Multiple source files.
- ○ Local labels.
- ○ Conditional assembly.
- ○ "Conditional jump" instructions become branches or branches plus jumps depending on distance.

□ Library The basic run-time library. These routines are used freely by all software.
- ○ Buffered character-by-character I/O.
- ○ Formatted input and output conversion (SCANF and PRINTF) for standard input and output, files, in-memory conversion.
- ○ Storage allocator.
- ○ Time conversions.
- ○ Number conversions.
- ○ Password encryption.
- ○ Quicksort.
- ○ Random number generator.
- ○ Mathematical function library, including trigonometric functions and inverses, exponential, logarithm, square root, bessel functions.

□ ADB — Interactive debugger.
- ○ Postmortem dumping.
- ○ Examination of arbitrary files, with no limit on size.
- ○ Interactive breakpoint debugging with the debugger as a separate process.
- ○ Symbolic reference to local and global variables.
- ○ Stack trace for C programs.
- ○ Output formats:
  - 1-, 2-, or 4-byte integers in octal, decimal, or hex
  - single and double floating point
  - character and string
  - disassembled machine instructions
- ○ Patching.
- ○ Searching for integer, character, or floating patterns.
- ○ Handles separated instruction and data space.

□ OD — Dump any file. Output options include any combination of octal or decimal by words, octal by bytes, ASCII, opcodes, hexadecimal.
- ○ Range of dumping is controllable.

□ LD — Link edit. Combine relocatable object files. Insert required routines from specified libraries.
- ○ Resulting code may be sharable.
- ○ Resulting code may have separate instruction and data spaces.

□ LORDER — Places object file names in proper order for loading, so that files depending on others come after them.

□ NM — Print the namelist (symbol table) of an object program. Provides control over the style and order of names that are printed.

□ SIZE — Report the core requirements of one or more object files.

□ STRIP — Remove the relocation and symbol table information from an object file to save space.

□ TIME — Run a command and report timing information on it.

□ PROF — Construct a profile of time spent per routine from statistics gathered by time-sampling the execution of a program. Uses floating point.
- ○ Subroutine call frequency and average times for C programs.

□ MAKE — Controls creation of large programs. Uses a control file specifying source file dependencies to make new version; uses time last changed to deduce minimum amount of work necessary.
- ○ Knows about CC, YACC, LEX, etc.

1.12. UNIX Programmer's Manual

□ Manual — Machine-readable version of the UNIX Programmer's Manual.
- ○ System overview.
- ○ All commands.
- ○ All system calls.
- ○ All subroutines in C and assembler libraries.
- ○ All devices and other special files.
- ○ Formats of file system and kinds of files known to system software.
- ○ Boot and maintenance procedures.

☐ MAN	Print specified manual section on your terminal.

1.13. Computer-Aided Instruction

☐ LEARN	A program for interpreting CAI scripts, plus scripts for learning about UNIX by using it.
○ Scripts for basic files and commands, editor, advanced files and commands, EQN, MS macros, C programming language.

2. Languages

2.1. The C Language

☐ CC	Compile and/or link edit programs in the C language. The UNIX operating system, most of the subsystems and C itself are written in C. For a full description of C, read *The C Programming Language*, Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1978.
○ General purpose language designed for structured programming.
○ Data types include character, integer, float, double, pointers to all types, functions returning above types, arrays of all types, structures and unions of all types.
○ Operations intended to give machine-independent control of full machine facility, including to-memory operations and pointer arithmetic.
○ Macro preprocessor for parameterized code and inclusion of standard files.
○ All procedures recursive, with parameters by value.
○ Machine-independent pointer manipulation.
○ Object code uses full addressing capability of the PDP-11.
○ Runtime library gives access to all system facilities.
○ Definable data types.
○ Block structure

☐ LINT	Verifier for C programs. Reports questionable or nonportable usage such as:
Mismatched data declarations and procedure interfaces.
Nonportable type conversions.
Unused variables, unreachable code, no-effect operations.
Mistyped pointers.
Obsolete syntax.
○ Full cross-module checking of separately compiled programs.

☐ CB	A beautifier for C programs. Does proper indentation and placement of braces.

2.2. Fortran

☐ F77	A full compiler for ANSI Standard Fortran 77.
○ Compatible with C and supporting tools at object level.
○ Optional source compatibility with Fortran 66.
○ Free format source.
○ Optional subscript-range checking, detection of uninitialized variables.
○ All widths of arithmetic: 2- and 4-byte integer; 4- and 8-byte real; 8- and 16-byte complex.

☐ RATFOR	Ratfor adds rational control structure à la C to Fortran.
○ Compound statements.

○ If-else, do, for, while, repeat-until, break, next statements.
○ Symbolic constants.
○ File insertion.
○ Free format source
○ Translation of relationals like >, >=.
○ Produces genuine Fortran to carry away.
○ May be used with F77.

□ STRUCT Converts ordinary ugly Fortran into structured Fortran (i.e., Ratfor), using statement grouping, if-else, while, for, repeat-until.

2.3. Other Algorithmic Languages

□ BAS An interactive interpreter, similar in style to BASIC. Interpret unnumbered statements immediately, numbered statements upon 'run'.
○ Statements include:
comment,
dump,
for...next,
goto,
if...else...fi,
list,
print,
prompt,
return,
run,
save.
○ All calculations double precision.
○ Recursive function defining and calling.
○ Builtin functions include log, exp, sin, cos, atn, int, sqr, abs, rnd.
○ Escape to ED for complex program editing.

□ DC Interactive programmable desk calculator. Has named storage locations as well as conventional stack for holding integers or programs.
○ Unlimited precision decimal arithmetic.
○ Appropriate treatment of decimal fractions.
○ Arbitrary input and output radices, in particular binary, octal, decimal and hexadecimal.
○ Reverse Polish operators:
+ − * /
remainder, power, square root,
load, store, duplicate, clear,
print, enter program text, execute.

□ BC A C-like interactive interface to the desk calculator DC.
○ All the capabilities of DC with a high-level syntax.
○ Arrays and recursive functions.
○ Immediate evaluation of expressions and evaluation of functions upon call.
○ Arbitrary precision elementary functions: exp, sin, cos, atan.
○ Go-to-less programming.

2.4. Macroprocessing

□ M4	A general purpose macroprocessor. ○ Stream-oriented, recognizes macros anywhere in text. ○ Syntax fits with functional syntax of most higher-level languages. ○ Can evaluate integer arithmetic expressions.

2.5. Compiler-compilers

□ YACC	An LR(1)-based compiler writing system. During execution of resulting parsers, arbitrary C functions may be called to do code generation or semantic actions. ○ BNF syntax specifications. ○ Precedence relations. ○ Accepts formally ambiguous grammars with non-BNF resolution rules.
□ LEX	Generator of lexical analyzers. Arbitrary C functions may be called upon isolation of each lexical token. ○ Full regular expression, plus left and right context dependence. ○ Resulting lexical analysers interface cleanly with YACC parsers.

3. Text Processing

3.1. Document Preparation

□ ED	Interactive context editor. Random access to all lines of a file. ○ Find lines by number or pattern. Patterns may include: specified characters, don't care characters, choices among characters, repetitions of these constructs, beginning of line, end of line. ○ Add, delete, change, copy, move or join lines. ○ Permute or split contents of a line. ○ Replace one or all instances of a pattern within a line. ○ Combine or split files. ○ Escape to Shell (command language) during editing. ○ Do any of above operations on every pattern-selected line in a given range. ○ Optional encryption for extra security.
□ PTX	Make a permuted (key word in context) index.
□ SPELL	Look for spelling errors by comparing each word in a document against a word list. ○ 25,000-word list includes proper names. ○ Handles common prefixes and suffixes. ○ Collects words to help tailor local spelling lists.
□ LOOK	Search for words in dictionary that begin with specified prefix.
□ TYPO	Look for spelling errors by a statistical technique; not limited to English.
□ CRYPT	Encrypt and decrypt files for security.

3.2. Document Formatting

□ ROFF	A typesetting program for terminals. Easy for nontechnical people to learn, and good for simple documents. Input consists of data lines intermixed with control lines, such as .sp 2 insert two lines of space .ce center the next line ROFF is deemed to be obsolete; it is intended only for casual use.

- ○ Justification of either or both margins.
- ○ Automatic hyphenation.
- ○ Generalized running heads and feet, with even-odd page capability, numbering, etc.
- ○ Definable macros for frequently used control sequences (no substitutable arguments).
- ○ All 4 margins and page size dynamically adjustable.
- ○ Hanging indents and one-line indents.
- ○ Absolute and relative parameter settings.
- ○ Optional legal-style numbering of output lines.
- ○ Multiple file capability.
- ○ Not usable as a filter.

□ TROFF

□ NROFF Advanced typesetting. TROFF drives a Graphic Systems phototypesetter; NROFF drives ascii terminals of all types. This summary was typeset using TROFF. TROFF and NROFF style is similar to ROFF, but they are capable of much more elaborate feats of formatting, when appropriately programmed. TROFF and NROFF accept the same input language.

- ○ All ROFF capabilities available or definable.
- ○ Completely definable page format keyed to dynamically planted "interrupts" at specified lines.
- ○ Maintains several separately definable typesetting environments (e.g., one for body text, one for footnotes, and one for unusually elaborate headings).
- ○ Arbitrary number of output pools can be combined at will.
- ○ Macros with substitutable arguments, and macros invocable in mid-line.
- ○ Computation and printing of numerical quantities.
- ○ Conditional execution of macros.
- ○ Tabular layout facility.
- ○ Positions expressible in inches, centimeters, ems, points, machine units or arithmetic combinations thereof.
- ○ Access to character-width computation for unusually difficult layout problems.
- ○ Overstrikes, built-up brackets, horizontal and vertical line drawing.
- ○ Dynamic relative or absolute positioning and size selection, globally or at the character level.
- ○ Can exploit the characteristics of the terminal being used, for approximating special characters, reverse motions, proportional spacing, etc.

The Graphic Systems typesetter has a vocabulary of several 102-character fonts (4 simultaneously) in 15 sizes. TROFF provides terminal output for rough sampling of the product.

NROFF will produce multicolumn output on terminals capable of reverse line feed, or through the postprocessor COL.

High programming skill is required to exploit the formatting capabilities of TROFF and NROFF, although unskilled personnel can easily be trained to enter documents according to canned formats such as those provided by MS, below. TROFF and EQN are essentially identical to NROFF and NEQN so it is usually possible to define interchangeable formats to produce approximate proof copy on terminals before actual typesetting. The preprocessors MS, TBL, and REFER are fully compatible with TROFF and NROFF.

□ MS A standardized manuscript layout package for use with NROFF/TROFF. This document was formatted with MS.

○ Page numbers and draft dates.
○ Automatically numbered subheads.
○ Footnotes.
○ Single or double column.
○ Paragraphing, display and indentation.
○ Numbered equations.

□ EQN — A mathematical typesetting preprocessor for TROFF. Translates easily readable formulas, either in-line or displayed, into detailed typesetting instructions. Formulas are written in a style like this:

sigma sup 2 ~=~ 1 over N sum from i=1 to N (x sub i − x bar) sup 2

which produces:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i-\bar{x})^2$$

○ Automatic calculation of size changes for subscripts, sub-subscripts, etc.
○ Full vocabulary of Greek letters and special symbols, such as 'gamma', 'GAMMA', 'integral'.
○ Automatic calculation of large bracket sizes.
○ Vertical "piling" of formulae for matrices, conditional alternatives, etc.
○ Integrals, sums, etc., with arbitrarily complex limits.
○ Diacriticals: dots, double dots, hats, bars, etc.
○ Easily learned by nonprogrammers and mathematical typists.

□ NEQN — A version of EQN for NROFF; accepts the same input language. Prepares formulas for display on any terminal that NROFF knows about, for example, those based on Diablo printing mechanism.
○ Same facilities as EQN within graphical capability of terminal.

□ TBL — A preprocessor for NROFF/TROFF that translates simple descriptions of table layouts and contents into detailed typesetting instructions.
○ Computes column widths.
○ Handles left- and right-justified columns, centered columns and decimal-point alignment.
○ Places column titles.
○ Table entries can be text, which is adjusted to fit.
○ Can box all or parts of table.

□ REFER — Fills in bibliographic citations in a document from a data base (not supplied).
○ References may be printed in any style, as they occur or collected at the end.
○ May be numbered sequentially, by name of author, etc.

□ TC — Simulate Graphic Systems typesetter on Tektronix 4014 scope. Useful for checking TROFF page layout before typesetting.

□ GREEK — Fancy printing on Diablo-mechanism terminals like DASI-300 and DASI-450, and on Tektronix 4014.
○ Gives half-line forward and reverse motions.
○ Approximates Greek letters and other special characters by overstriking.

□ COL — Canonicalize files with reverse line feeds for one-pass printing.

□ DEROFF — Remove all TROFF commands from input.

□ CHECKEQ — Check document for possible errors in EQN usage.

4. Information Handling

◻ SORT	Sort or merge ASCII files line-by-line. No limit on input size. ○ Sort up or down. ○ Sort lexicographically or on numeric key. ○ Multiple keys located by delimiters or by character position. ○ May sort upper case together with lower into dictionary order. ○ Optionally suppress duplicate data.
◻ TSORT	Topological sort — converts a partial order into a total order.
◻ UNIQ	Collapse successive duplicate lines in a file into one line. ○ Publish lines that were originally unique, duplicated, or both. ○ May give redundancy count for each line.
◻ TR	Do one-to-one character translation according to an arbitrary code. ○ May coalesce selected repeated characters. ○ May delete selected characters.
◻ DIFF	Report line changes, additions and deletions necessary to bring two files into agreement. ○ May produce an editor script to convert one file into another. ○ A variant compares two new versions against one old one.
◻ COMM	Identify common lines in two sorted files. Output in up to 3 columns shows lines present in first file only, present in both, and/or present in second only.
◻ JOIN	Combine two files by joining records that have identical keys.
◻ GREP	Print all lines in a file that satisfy a pattern as used in the editor ED. ○ May print all lines that fail to match. ○ May print count of hits. ○ May print first hit in each file.
◻ LOOK	Binary search in sorted file for lines with specified prefix.
◻ WC	Count the lines, "words" (blank-separated strings) and characters in a file.
◻ SED	Stream-oriented version of ED. Can perform a sequence of editing operations on each line of an input stream of unbounded length. ○ Lines may be selected by address or range of addresses. ○ Control flow and conditional testing. ○ Multiple output streams. ○ Multi-line capability.
◻ AWK	Pattern scanning and processing language. Searches input for patterns, and performs actions on each line of input that satisfies the pattern. ○ Patterns include regular expressions, arithmetic and lexicographic conditions, boolean combinations and ranges of these. ○ Data treated as string or numeric as appropriate. ○ Can break input into fields; fields are variables. ○ Variables and arrays (with non-numeric subscripts). ○ Full set of arithmetic operators and control flow. ○ Multiple output streams to files and pipes. ○ Output can be formatted as desired. ○ Multi-line capabilities.

5. Graphics

The programs in this section are predominantly intended for use with Tektronix 4014 storage scopes.

□ GRAPH	Prepares a graph of a set of input numbers. ○ Input scaled to fit standard plotting area. ○ Abscissae may be supplied automatically. ○ Graph may be labeled. ○ Control over grid style, line style, graph orientation, etc.
□ SPLINE	Provides a smooth curve through a set of points intended for GRAPH.
□ PLOT	A set of filters for printing graphs produced by GRAPH and other programs on various terminals. Filters provided for 4014, DASI terminals, Versatec printer/plotter.

6. Novelties, Games, and Things That Didn't Fit Anywhere Else

□ BACKGAMMON	A player of modest accomplishment.
□ CHESS	Plays good class D chess.
□ CHECKERS	Ditto, for checkers.
□ BCD	Converts ascii to card-image form.
□ PPT	Converts ascii to paper tape form.
□ BJ	A blackjack dealer.
□ CUBIC	An accomplished player of 4×4×4 tic-tac-toe.
□ MAZE	Constructs random mazes for you to solve.
□ MOO	A fascinating number-guessing game.
□ CAL	Print a calendar of specified month and year.
□ BANNER	Print output in huge letters.
□ CHING	The *I Ching*. Place your own interpretation on the output.
□ FORTUNE	Presents a random fortune cookie on each invocation. Limited jar of cookies included.
□ UNITS	Convert amounts between different scales of measurement. Knows hundreds of units. For example, how many km/sec is a parsec/megayear?
□ TTT	A tic-tac-toe program that learns. It never makes the same mistake twice.
□ ARITHMETIC	Speed and accuracy test for number facts.
□ FACTOR	Factor large integers.
□ QUIZ	Test your knowledge of Shakespeare, Presidents, capitals, etc.
□ WUMP	Hunt the wumpus, thrilling search in a dangerous cave.
□ REVERSI	A two person board game, isomorphic to Othello®.
□ HANGMAN	Word-guessing game. Uses the dictionary supplied with SPELL.

☐ FISH	Children's card-guessing game.

The UNIX Time-Sharing System*

D. M. Ritchie and K. Thompson

ABSTRACT

UNIX† is a general-purpose, multi-user, interactive operating system for the larger Digital Equipment Corporation PDP-11 and the Interdata 8/32 computers. It offers a number of features seldom found even in larger operating systems, including

i A hierarchical file system incorporating demountable volumes,

ii Compatible file, device, and inter-process I/O,

iii The ability to initiate asynchronous processes,

iv System command language selectable on a per-user basis,

v Over 100 subsystems including a dozen languages,

vi High degree of portability.

This paper discusses the nature and implementation of the file system and of the user command interface.

1. INTRODUCTION

There have been four versions of the UNIX time-sharing system. The earliest (circa 1969-70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. The third incorporated multiprogramming and ran on the PDP-11/34, /40, /45, /60, and /70 computers; it is the one described in the previously published version of this paper, and is also the most widely used today. This paper describes only the fourth, current system that runs on the PDP-11/70 and the Interdata 8/32 computers. In fact, the differences among the various systems is rather small; most of the revisions made to the originally published version of this paper, aside from those concerned with style, had to do with details of the implementation of the file system.

Since PDP-11 UNIX became operational in February, 1971, over 600 installations have been put into service. Most of them are engaged in applications such as computer science education, the preparation and formatting of documents and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: it can run on hardware costing as little as $40,000, and less than two man-years were spent on the main system software. We hope, however, that users find that the most important

* Copyright 1974, Association for Computing Machinery, Inc., reprinted by permission. This is a revised version of an article that appeared in Communications of the ACM, *17*, No. 7 (July 1974), pp. 365-375. That article was a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973.

†UNIX is a Trademark of Bell Laboratories.

characteristics of the system are its simplicity, elegance, and ease of use.

Besides the operating system proper, some major programs available under UNIX are

C compiler
Text editor based on QED[1]
Assembler, linking loader, symbolic debugger
Phototypesetting and equation setting programs[2, 3]
Dozens of languages including Fortran 77, Basic, Snobol, APL, Algol 68, M6, TMG, Pascal

There is a host of maintenance, utility, recreation and novelty programs, all written locally. The UNIX user community, which numbers in the thousands, has contributed many more programs and languages. It is worth noting that the system is totally self-supporting. All UNIX software is maintained on the system; likewise, this paper and all other documents in this issue were generated and formatted by the UNIX editor and text formatting programs.

II. HARDWARE AND SOFTWARE ENVIRONMENT

The PDP-11/70 on which the Research UNIX system is installed is a 16-bit word (8-bit byte) computer with 768K bytes of core memory; the system kernel occupies 90K bytes about equally divided between code and data tables. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system capable of running the software mentioned above can require as little as 96K bytes of core altogether. There are even larger installations; see the description of the PWB/UNIX systems,[4, 5] for example. There are also much smaller, though somewhat restricted, versions of the system.[6]

Our own PDP-11 has two 200-Mb moving-head disks for file system storage and swapping. There are 20 variable-speed communications interfaces attached to 300- and 1200-baud data sets, and an additional 12 communication lines hard-wired to 9600-baud terminals and satellite computers. There are also several 2400- and 4800-baud synchronous communication interfaces used for machine-to-machine file transfer. Finally, there is a variety of miscellaneous devices including nine-track magnetic tape, a line printer, a voice synthesizer, a phototypesetter, a digital switching network, and a chess machine.

The preponderance of UNIX software is written in the abovementioned C language.[7] Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system was about one-third greater than that of the old. Since the new system not only became much easier to understand and to modify but also included many functional improvements, including multiprogramming and the ability to share reentrant code among several user programs, we consider this increase in size quite acceptable.

III. THE FILE SYSTEM

The most important role of the system is to provide a file system. From the point of view of the user, there are three kinds of files: ordinary disk files, directories, and special files.

3.1 Ordinary files

A file contains whatever information the user places on it, for example, symbolic or binary (object) programs. No particular structuring is expected by the system. A file of text consists simply of a string of characters, with lines demarcated by the newline character. Binary programs are sequences of words as they will appear in core memory when the program starts executing. A few user programs manipulate files with more structure; for example, the assembler generates, and the loader expects, an object file in a particular format. However, the structure of files is controlled by the programs that use them, not by the system.

3.2 Directories

Directories provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole. Each user has a directory of his own files; he may also create subdirectories to contain groups of files conveniently treated together. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs, so that the system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

The system maintains several directories for its own use. One of these is the **root** directory. All files in the system can be found by tracing a path through a chain of directories until the desired file is reached. The starting point for such searches is often the **root**. Other system directories contain all the programs provided for general use; that is, all the *commands*. As will be seen, however, it is by no means necessary that a program reside in one of these directories for it to be executed.

Files are named by sequences of 14 or fewer characters. When the name of a file is specified to the system, it may be in the form of a *path name*, which is a sequence of directory names separated by slashes, "/", and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. The name **/alpha/beta/gamma** causes the system to search the root for directory **alpha**, then to search **alpha** for **beta**, finally to find **gamma** in **beta**. **gamma** may be an ordinary file, a directory, or a special file. As a limiting case, the name "/" refers to the root itself.

A path name not starting with "/" causes the system to begin the search in the user's current directory. Thus, the name **alpha/beta** specifies the file named **beta** in subdirectory **alpha** of the current directory. The simplest kind of name, for example, **alpha**, refers to a file that itself is found in the current directory. As another limiting case, the null file name refers to the current directory.

The same non-directory file may appear in several directories under possibly different names. This feature is called *linking*; a directory entry for a file is sometimes called a link. The UNIX system differs from other systems in which linking is permitted in that all links to a file have equal status. That is, a file does not exist within a particular directory; the directory entry for a file consists merely of its name and a pointer to the information actually describing the file. Thus a file exists independently of any directory entry, although in practice a file is made to disappear along with the last link to it.

Each directory always has at least two entries. The name "." in each directory refers to the directory itself. Thus a program may read the current directory under the name "." without knowing its complete path name. The name ".." by convention refers to the parent of the directory in which it appears, that is, to the directory in which it was created.

The directory structure is constrained to have the form of a rooted tree. Except for the special entries "." and "..", each directory must appear as an entry in exactly one other directory, which is its parent. The reason for this is to simplify the writing of programs that visit subtrees of the directory structure, and more important, to avoid the separation of portions of the hierarchy. If arbitrary links to directories were permitted, it would be quite difficult to detect when the last connection from the root to a directory was severed.

3.3 Special files

Special files constitute the most unusual feature of the UNIX file system. Each supported I/O device is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory **/dev**, although a link may be made to one of these files just as it may to an ordinary file. Thus, for example, to write on a magnetic tape one may write on the file **/dev/mt**. Special files exist for each communication line, each disk, each tape drive, and for physical main memory. Of course, the active disks and the memory special file are protected from indiscriminate access.

There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.

3.4 Removable file systems

Although the root of the file system is always stored on the same device, it is not necessary that the entire file system hierarchy reside on this device. There is a **mount** system request with two arguments: the name of an existing ordinary file, and the name of a special file whose associated storage volume (e.g., a disk pack) should have the structure of an independent file system containing its own directory hierarchy. The effect of **mount** is to cause references to the heretofore ordinary file to refer instead to the root directory of the file system on the removable volume. In effect, **mount** replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume). After the **mount**, there is virtually no distinction between files on the removable volume and those in the permanent file system. In our installation, for example, the root directory resides on a small partition of one of our disk drives, while the other drive, which contains the user's files, is mounted by the system initialization sequence. A mountable file system is generated by writing on its corresponding special file. A utility program is available to create an empty file system, or one may simply copy an existing file system.

There is only one exception to the rule of identical treatment of files on different devices: no link may exist between one file system hierarchy and another. This restriction is enforced so as to avoid the elaborate bookkeeping that would otherwise be required to assure removal of the links whenever the removable volume is dismounted.

3.5 Protection

Although the access control scheme is quite simple, it has some unusual features. Each user of the system is assigned a unique user identification number. When a file is created, it is marked with the user ID of its owner. Also given for new files is a set of ten protection bits. Nine of these specify independently read, write, and execute permission for the owner of the file, for other members of his group, and for all remaining users.

If the tenth bit is on, the system will temporarily change the user identification (hereafter, user ID) of the current user to that of the creator of the file whenever the file is executed as a program. This change in user ID is effective only during the execution of the program that calls for it. The set-user-ID feature provides for privileged programs that may use files inaccessible to other users. For example, a program may keep an accounting file that should neither be read nor changed except by the program itself. If the set-user-ID bit is on for the program, it may access the file although this access might be forbidden to other programs invoked by the given program's user. Since the actual user ID of the invoker of any program is always available, set-user-ID programs may take any measures desired to satisfy themselves as to their invoker's credentials. This mechanism is used to allow users to execute the carefully written commands that call privileged system entries. For example, there is a system entry invokable only by the "super-user" (below) that creates an empty directory. As indicated above, directories are expected to have entries for "." and "..". The command which creates a directory is owned by the super-user and has the set-user-ID bit set. After it checks its invoker's authorization to create the specified directory, it creates it and makes the entries for "." and "..".

Because anyone may set the set-user-ID bit on one of his own files, this mechanism is generally available without administrative intervention. For example, this protection scheme easily solves the MOO accounting problem posed by "Aleph-null."[8]

The system recognizes one particular user ID (that of the "super-user") as exempt from the usual constraints on file access; thus (for example), programs may be written to dump and reload the file system without unwanted interference from the protection system.

3.6 I/O calls

The system calls to do I/O are designed to eliminate the differences between the various devices and styles of access. There is no distinction between "random" and "sequential" I/O, nor is any logical record size imposed by the system. The size of an ordinary file is determined by the number of bytes written on it; no predetermination of the size of a file is necessary or possible.

To illustrate the essentials of I/O, some of the basic calls are summarized below in an anonymous language that will indicate the required parameters without getting into the underlying complexities. Each call to the system may potentially result in an error return, which for simplicity is not represented in the calling sequence.

To read or write a file assumed to exist already, it must be opened by the following call:

```
filep = open(name, flag)
```

where **name** indicates the name of the file. An arbitrary path name may be given. The **flag** argument indicates whether the file is to be read, written, or "updated," that is, read and written simultaneously.

The returned value **filep** is called a *file descriptor*. It is a small integer used to identify the file in subsequent calls to read, write, or otherwise manipulate the file.

To create a new file or completely rewrite an old one, there is a **create** system call that creates the given file if it does not exist, or truncates it to zero length if it does exist; **create** also opens the new file for writing and, like **open**, returns a file descriptor.

The file system maintains no locks visible to the user, nor is there any restriction on the number of users who may have a file open for reading or writing. Although it is possible for the contents of a file to become scrambled when two users write on it simultaneously, in practice difficulties do not arise. We take the view that locks are neither necessary nor sufficient, in our environment, to prevent interference between users of the same file. They are unnecessary because we are not faced with large, single-file data bases maintained by independent processes. They are insufficient because locks in the ordinary sense, whereby one user is prevented from writing on a file that another user is reading, cannot prevent confusion when, for example, both users are editing a file with an editor that makes a copy of the file being edited.

There are, however, sufficient internal interlocks to maintain the logical consistency of the file system when two users engage simultaneously in activities such as writing on the same file, creating files in the same directory, or deleting each other's open files.

Except as indicated below, reading and writing are sequential. This means that if a particular byte in the file was the last byte written (or read), the next I/O call implicitly refers to the immediately following byte. For each open file there is a pointer, maintained inside the system, that indicates the next byte to be read or written. If *n* bytes are read or written, the pointer advances by *n* bytes.

Once a file is open, the following calls may be used:

```
n = read(filep, buffer, count)
n = write(filep, buffer, count)
```

Up to **count** bytes are transmitted between the file specified by **filep** and the byte array specified by **buffer**. The returned value **n** is the number of bytes actually transmitted. In the **write** case, **n** is the same as **count** except under exceptional conditions, such as I/O errors or end of physical medium on special files; in a **read**, however, **n** may without error be less than **count**. If the read pointer is so near the end of the file that reading **count** characters would cause reading beyond the end, only sufficient bytes are transmitted to reach the end of the file; also, typewriter-like terminals never return more than one line of input. When a **read** call returns with **n** equal to zero, the end of the file has been reached. For disk files this occurs when the read pointer becomes equal to the current size of the file. It is possible to generate an end-of-file from a terminal by use of an escape sequence that depends on the device used.

Bytes written affect only those parts of a file implied by the position of the write pointer and the count; no other part of the file is changed. If the last byte lies beyond the end of the file, the file is made to grow as needed.

To do random (direct-access) I/O it is only necessary to move the read or write pointer to the appropriate location in the file.

```
location = lseek ( filep, offset, base )
```

The pointer associated with **filep** is moved to a position **offset** bytes from the beginning of the file, from the current position of the pointer, or from the end of the file, depending on **base**. **offset** may be negative. For some devices (e.g., paper tape and terminals) seek calls are ignored. The actual offset from the beginning of the file to which the pointer was moved is returned in **location**.

There are several additional system entries having to do with I/O and with the file system that will not be discussed. For example: close a file, get the status of a file, change the protection mode or the owner of a file, create a directory, make a link to an existing file, delete a file.

IV. IMPLEMENTATION OF THE FILE SYSTEM

As mentioned in Section 3.2 above, a directory entry contains only a name for the associated file and a pointer to the file itself. This pointer is an integer called the *i-number* (for index number) of the file. When the file is accessed, its i-number is used as an index into a system table (the *i-list*) stored in a known part of the device on which the directory resides. The entry found thereby (the file's *i-node*) contains the description of the file:

i	the user and group-ID of its owner
ii	its protection bits
iii	the physical disk or tape addresses for the file contents
iv	its size
v	time of creation, last use, and last modification
vi	the number of links to the file, that is, the number of times it appears in a directory
vii	a code indicating whether the file is a directory, an ordinary file, or a special file.

The purpose of an **open** or **create** system call is to turn the path name given by the user into an i-number by searching the explicitly or implicitly named directories. Once a file is open, its device, i-number, and read/write pointer are stored in a system table indexed by the file descriptor returned by the **open** or **create**. Thus, during a subsequent call to read or write the file, the descriptor may be easily related to the information necessary to access the file.

When a new file is created, an i-node is allocated for it and a directory entry is made that contains the name of the file and the i-node number. Making a link to an existing file involves creating a directory entry with the new name, copying the i-number from the original file entry, and incrementing the link-count field of the i-node. Removing (deleting) a file is done by decrementing the link-count of the i-node specified by its directory entry and erasing the directory entry. If the link-count drops to 0, any disk blocks in the file are freed and the i-node is de-allocated.

The space on all disks that contain a file system is divided into a number of 512-byte blocks logically addressed from 0 up to a limit that depends on the device. There is space in the i-node of each file for 13 device addresses. For nonspecial files, the first 10 device addresses point at the first 10 blocks of the file. If the file is larger than 10 blocks, the 11 device address points to an indirect block containing up to 128 addresses of additional blocks in the file. Still larger files use the twelfth device address of the i-node to point to a double-indirect block naming 128 indirect blocks, each pointing to 128 blocks of the file. If required, the thirteenth device address is a triple-indirect block. Thus files may conceptually grow to $[(10+128+128^2+128^3)\cdot 512]$ bytes. Once opened, bytes numbered below 5120 can be read with a single disk access; bytes in the range 5120 to 70,656 require two accesses; bytes in the range 70,656 to 8,459,264 require three accesses; bytes from there to the largest file (1,082,201,088) require four accesses. In practice, a device cache mechanism (see below) proves effective in eliminating most of the indirect fetches.

The foregoing discussion applies to ordinary files. When an I/O request is made to a file whose i-node indicates that it is special, the last 12 device address words are immaterial, and the first specifies an internal *device name*, which is interpreted as a pair of numbers representing, respectively, a device type and subdevice number. The device type indicates which system routine will deal with I/O on that device; the subdevice number selects, for example, a disk drive attached to a particular controller or one of several similar terminal interfaces.

In this environment, the implementation of the **mount** system call (Section 3.4) is quite straightforward. **mount** maintains a system table whose argument is the i-number and device name of the ordinary file specified during the **mount**, and whose corresponding value is the device name of the indicated special file. This table is searched for each i-number/device pair that turns up while a path name is being scanned during an **open** or **create**; if a match is found, the i-number is replaced by the i-number of the root directory and the device name is replaced by the table value.

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is, immediately after return from a **read** call the data are available; conversely, after a **write** the user's workspace may be reused. In fact, the system maintains a rather complicated buffering mechanism that reduces greatly the number of I/O operations required to access a file. Suppose a **write** call is made specifying transmission of a single byte. The system will search its buffers to see whether the affected disk block currently resides in main memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer and an entry is made in a list of blocks to be written. The return from the **write** call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

The system recognizes when a program has made accesses to sequential blocks of a file, and asynchronously pre-reads the next block. This significantly reduces the running time of most programs while adding little to system overhead.

A program that reads or writes files in units of 512 bytes has an advantage over a program that reads or writes a single byte at a time, but the gain is not immense; it comes mainly from the avoidance of system overhead. If a program is used rarely or does no great volume of I/O, it may quite reasonably read and write in units as small as it wishes.

The notion of the i-list is an unusual feature of UNIX. In practice, this method of organizing the file system has proved quite reliable and easy to deal with. To the system itself, one of its strengths is the fact that each file has a short, unambiguous name related in a simple way to the protection, addressing, and other information needed to access the file. It also permits a quite simple and rapid algorithm for checking the consistency of a file system, for example, verification that the portions of each device containing useful information and those free to be allocated are disjoint and together exhaust the space on the device. This algorithm is independent of the directory hierarchy, because it need only scan the linearly organized i-list. At the same time the notion of the i-list induces certain peculiarities not found in other file system organizations. For example, there is the question of who is to be charged for the space a file occupies, because all directory entries for a file have equal status. Charging the owner of a file is unfair in general, for one user may create a file, another may link to it, and the first user may delete the file. The first user is still the owner of the file, but it should be charged to the second user. The simplest reasonably fair algorithm seems to be to spread the charges equally among users who have links to a file. Many installations avoid the issue by not charging any fees at all.

V. PROCESSES AND IMAGES

An *image* is a computer execution environment. It includes a memory image, general register values, status of open files, current directory and the like. An image is the current state of a pseudo-computer.

A *process* is the execution of an image. While the processor is executing on behalf of a process, the image must reside in main memory; during the execution of other processes it remains in main memory unless the appearance of an active, higher-priority process forces it to be swapped out to the disk.

The user-memory part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first hardware protection byte boundary above the program text segment in the virtual address space begins a non-shared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the stack pointer fluctuates.

5.1 Processes

Except while the system is bootstrapping itself into operation, a new process can come into existence only by use of the **fork** system call:

processid = fork ( )

When **fork** is executed, the process splits into two independently executing processes. The two processes have independent copies of the original memory image, and share all open files. The new processes differ only in that one is considered the parent process: in the parent, the returned **processid** actually identifies the child process and is never 0, while in the child, the returned value is always 0.

Because the values returned by **fork** in the parent and child process are distinguishable, each process may determine whether it is the parent or child.

5.2 Pipes

Processes may communicate with related processes using the same system **read** and **write** calls that are used for file-system I/O. The call:

filep = pipe ( )

returns a file descriptor **filep** and creates an inter-process channel called a *pipe*. This channel, like other open files, is passed from parent to child process in the image by the **fork** call. A **read** using a pipe file descriptor waits until another process writes using the file descriptor for the same pipe. At this point, data are passed between the images of the two processes. Neither process need know that a pipe, rather than an ordinary file, is involved.

Although inter-process communication via pipes is a quite valuable tool (see Section 6.2), it is not a completely general mechanism, because the pipe must be set up by a common ancestor of the processes involved.

5.3 Execution of programs

Another major system primitive is invoked by

execute (file, arg_1, arg_2, . . . , arg_n)

which requests the system to read in and execute the program named by **file**, passing it string arguments $\textbf{arg}_1$, $\textbf{arg}_2$, . . . , $\textbf{arg}_n$. All the code and data in the process invoking **execute** is replaced from the **file**, but open files, current directory, and inter-process relationships are unaltered. Only if the call fails, for example because **file** could not be found or because its execute-permission bit was not set, does a return take place from the **execute** primitive; it resembles a "jump" machine instruction rather than a subroutine call.

5.4 Process synchronization

Another process control system call:

```
processid = wait (status)
```

causes its caller to suspend execution until one of its children has completed execution. Then **wait** returns the **processid** of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available.

5.5 Termination

Lastly:

```
exit (status)
```

terminates a process, destroys its image, closes its open files, and generally obliterates it. The parent is notified through the **wait** primitive, and **status** is made available to it. Processes may also terminate as a result of various illegal actions or user-generated signals (Section VII below).

VI. THE SHELL

For most users, communication with the system is carried on with the aid of a program called the shell. The shell is a command-line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs. (The shell is described fully elsewhere,[9] so this section will discuss only the theory of its operation.) In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

command arg_1 arg_2 ... arg_n

The shell splits up the command name and the arguments into separate strings. Then a file with name **command** is sought; **command** may be a path name including the "/" character to specify any file in the system. If **command** is found, it is brought into memory and executed. The arguments collected by the shell are accessible to the command. When the command is finished, the shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file **command** cannot be found, the shell generally prefixes a string such as **/bin/** to **command** and attempts again to find the file. Directory **/bin** contains commands intended to be generally used. (The sequence of directories to be searched may be changed by user request.)

6.1 Standard I/O

The discussion of I/O in Section III above seems to imply that every file used by a program must be opened or created by the program in order to get a file descriptor for the file. Programs executed by the shell, however, start off with three open files with file descriptors 0, 1, and 2. As such a program begins execution, file 1 is open for writing, and is best understood as the standard output file. Except under circumstances indicated below, this file is the user's terminal. Thus programs that wish to write informative information ordinarily use file descriptor 1. Conversely, file 0 starts off open for reading, and programs that wish to read messages typed by the user read this file.

The shell is able to change the standard assignments of these file descriptors from the user's terminal printer and keyboard. If one of the arguments to a command is prefixed by ">", file descriptor 1 will, for the duration of the command, refer to the file named after the ">". For example:

```
ls
```

ordinarily lists, on the typewriter, the names of the files in the current directory. The command:

```
ls >there
```

creates a file called **there** and places the listing there. Thus the argument **>there** means "place output on **there**." On the other hand:

```
ed
```

ordinarily enters the editor, which takes requests from the user via his keyboard. The command

```
ed <script
```

interprets **script** as a file of editor commands; thus **<script** means "take input from **script**."

Although the file name following "<" or ">" appears to be an argument to the command, in fact it is interpreted completely by the shell and is not passed to the command at all. Thus no special coding to handle I/O redirection is needed within each command; the command need merely use the standard file descriptors 0 and 1 where appropriate.

File descriptor 2 is, like file 1, ordinarily associated with the terminal output stream. When an output-diversion request with ">" is specified, file 2 remains attached to the terminal, so that commands may produce diagnostic messages that do not silently end up in the output file.

6.2 Filters

An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the shell to execute all the commands simultaneously and to arrange that the standard output of each command be delivered to the standard input of the next command in the sequence. Thus in the command line:

```
ls | pr -2 | opr
```

ls lists the names of the files in the current directory; its output is passed to **pr**, which paginates its input with dated headings. (The argument "−2" requests double-column output.) Likewise, the output from **pr** is input to **opr**; this command spools its input onto a file for off-line printing.

This procedure could have been carried out more clumsily by:

```
ls >temp1
pr -2 <temp1 >temp2
opr <temp2
```

followed by removal of the temporary files. In the absence of the ability to redirect output and input, a still clumsier method would have been to require the **ls** command to accept user requests to paginate its output, to print in multi-column format, and to arrange that its output be delivered off-line. Actually it would be surprising, and in fact unwise for efficiency reasons, to expect authors of commands such as **ls** to provide such a wide variety of output options.

A program such as **pr** which copies its standard input to its standard output (with processing) is called a *filter*. Some filters that we have found useful perform character transliteration, selection of lines according to a pattern, sorting of the input, and encryption and decryption.

6.3 Command separators; multitasking

Another feature provided by the shell is relatively straightforward. Commands need not be on different lines; instead they may be separated by semicolons:

```
ls; ed
```

will first list the contents of the current directory, then enter the editor.

A related feature is more interesting. If a command is followed by "&," the shell will not wait for the command to finish before prompting again; instead, it is ready immediately to accept a new command. For example:

```
as source >output &
```

causes **source** to be assembled, with diagnostic output going to **output**; no matter how long the assembly takes, the shell returns immediately. When the shell does not wait for the completion of a command, the identification number of the process running that command is printed. This identification may be used to wait for the completion of the command or to terminate it. The "&" may be used several times in a line:

```
as source >output & ls >files &
```

does both the assembly and the listing in the background. In these examples, an output file other than the terminal was provided; if this had not been done, the outputs of the various commands would have been intermingled.

The shell also allows parentheses in the above operations. For example:

```
(date; ls) >x &
```

writes the current date and time followed by a list of the current directory onto the file **x**. The shell also returns immediately for another request.

6.4 The shell as a command; command files

The shell is itself a command, and may be called recursively. Suppose file **tryout** contains the lines:

```
as source
mv a.out testprog
testprog
```

The **mv** command causes the file **a.out** to be renamed **testprog**. **a.out** is the (binary) output of the assembler, ready to be executed. Thus if the three lines above were typed on the keyboard, **source** would be assembled, the resulting program renamed **testprog**, and **testprog** executed. When the lines are in **tryout**, the command:

```
sh <tryout
```

would cause the shell **sh** to execute the commands sequentially.

The shell has further capabilities, including the ability to substitute parameters and to construct argument lists from a specified subset of the file names in a directory. It also provides general conditional and looping constructions.

6.5 Implementation of the shell

The outline of the operation of the shell can now be understood. Most of the time, the shell is waiting for the user to type a command. When the newline character ending the line is typed, the shell's **read** call returns. The shell analyzes the command line, putting the arguments in a form appropriate for **execute**. Then **fork** is called. The child process, whose code of course is still that of the shell, attempts to perform an **execute** with the appropriate arguments. If successful, this will bring in and start execution of the program whose name was given. Meanwhile, the other process resulting from the **fork**, which is the parent process, **waits** for the child process to die. When this happens, the shell knows the command is finished, so it types its prompt and reads the keyboard to obtain another command.

Given this framework, the implementation of background processes is trivial: whenever a command line contains "&," the shell merely refrains from waiting for the process that it created to execute the command.

Happily, all of this mechanism meshes very nicely with the notion of standard input and output files. When a process is created by the **fork** primitive, it inherits not only the memory image of its parent but also all the files currently open in its parent, including those with file descriptors 0, 1, and 2. The shell, of course, uses these files to read command lines and to write its prompts and diagnostics, and in the ordinary case its children—the command programs—inherit them automatically. When an argument with "<" or ">" is given, however, the offspring process, just before it performs **execute**, makes the standard I/O file descriptor (0 or 1, respectively) refer to the named file. This is easy because, by agreement, the smallest unused file descriptor is assigned when a new file is opened (or **created**); it is only necessary to close file 0 (or 1) and open the named file. Because the process in which the command program runs simply terminates when it is through, the association between a file specified after "<" or ">" and file descriptor 0 or 1 is ended automatically when the process dies. Therefore the shell need not know the actual names of the files that are its own standard input and output, because it need never reopen them.

Filters are straightforward extensions of standard I/O redirection with pipes used instead of files.

In ordinary circumstances, the main loop of the shell never terminates. (The main loop includes the branch of the return from **fork** belonging to the parent process; that is, the branch that does a **wait**, then reads another command line.) The one thing that causes the shell to terminate is discovering an end-of-file condition on its input file. Thus, when the shell is executed as a command with a given input file, as in:

```
sh <comfile
```

the commands in **comfile** will be executed until the end of **comfile** is reached; then the instance of the shell invoked by **sh** will terminate. Because this shell process is the child of another instance of the shell, the **wait** executed in the latter will return, and another command may then be processed.

6.6 Initialization

The instances of the shell to which users type commands are themselves children of another process. The last step in the initialization of the system is the creation of a single process and the invocation (via **execute**) of a program called **init**. The role of **init** is to create one process for each terminal channel. The various subinstances of **init** open the appropriate terminals for input and output on files 0, 1, and 2, waiting, if necessary, for carrier to be established on dial-up lines. Then a message is typed out requesting that the user log in. When the user types a name or other identification, the appropriate instance of **init** wakes up, receives the log-in line, and reads a password file. If the user's name is found, and if he is able to supply the correct password, **init** changes to the user's default current directory, sets the process's user ID to that of the person logging in, and performs an **execute** of the shell. At this point, the shell is ready to receive commands and the logging-in protocol is complete.

Meanwhile, the mainstream path of **init** (the parent of all the subinstances of itself that will later become shells) does a **wait**. If one of the child processes terminates, either because a shell found an end of file or because a user typed an incorrect name or password, this path of **init** simply recreates the defunct process, which in turn reopens the appropriate input and output files and types another log-in message. Thus a user may log out simply by typing the end-of-file sequence to the shell.

6.7 Other programs as shell

The shell as described above is designed to allow users full access to the facilities of the system, because it will invoke the execution of any program with appropriate protection mode. Sometimes, however, a different interface to the system is desirable, and this feature is easily arranged for.

Recall that after a user has successfully logged in by supplying a name and password, **init** ordinarily invokes the shell to interpret command lines. The user's entry in the password file may contain the name of a program to be invoked after log-in instead of the shell. This program is free to interpret the user's messages in any way it wishes.

For example, the password file entries for users of a secretarial editing system might specify that the editor **ed** is to be used instead of the shell. Thus when users of the editing system log in, they are inside the editor and can begin work immediately; also, they can be prevented from invoking programs not intended for their use. In practice, it has proved desirable to allow a temporary escape from the editor to execute the formatting program and other utilities.

Several of the games (e.g., chess, blackjack, 3D tic-tac-toe) available on the system illustrate a much more severely restricted environment. For each of these, an entry exists in the password file specifying that the appropriate game-playing program is to be invoked instead of the shell. People who log in as a player of one of these games find themselves limited to the game and unable to investigate the (presumably more interesting) offerings of the UNIX system as a whole.

VII. TRAPS

The PDP-11 hardware detects a number of program faults, such as references to non-existent memory, unimplemented instructions, and odd addresses used where an even address is required. Such faults cause the processor to trap to a system routine. Unless other arrangements have been made, an illegal action causes the system to terminate the process and to write its image on file **core** in the current directory. A debugger can be used to determine the state of the program at the time of the fault.

Programs that are looping, that produce unwanted output, or about which the user has second thoughts may be halted by the use of the **interrupt** signal, which is generated by typing the "delete" character. Unless special action has been taken, this signal simply causes the program to cease execution without producing a **core** file. There is also a **quit** signal used to force an image file to be produced. Thus programs that loop unexpectedly may be halted and the remains inspected without prearrangement.

The hardware-generated faults and the interrupt and quit signals can, by request, be either ignored or caught by a process. For example, the shell ignores quits to prevent a quit from logging the user out. The editor catches interrupts and returns to its command level. This is useful for stopping long printouts without losing work in progress (the editor manipulates a copy of the file it is editing). In systems without floating-point hardware, unimplemented instructions are caught and floating-point instructions are interpreted.

VIII. PERSPECTIVE

Perhaps paradoxically, the success of the UNIX system is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer facilities, discovered a little-used PDP-7 and set out to create a more hospitable environment. This (essentially personal) effort was sufficiently successful to gain the interest of the other author and several colleagues, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. When in turn the 11/20 was outgrown, the system had proved useful enough to persuade management to invest in the PDP-11/45, and later in the PDP-11/70 and Interdata 8/32 machines, upon which it developed to its present form. Our goals throughout the effort, when articulated at all, have always been to build a comfortable relationship with the machine and to explore ideas and inventions in operating systems and other software. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Three considerations that influenced the design of UNIX are visible in retrospect.

First: because we are programmers, we naturally designed the system to make it easy to write, test, and run programs. The most important expression of our desire for programming convenience was that the system was arranged for interactive use, even though the original version only supported one user. We believe that a properly designed interactive system is much more productive and satisfying to use than a "batch" system. Moreover, such a system is rather easily adaptable to noninteractive use, while the converse is not true.

Second: there have always been fairly severe size constraints on the system and its software. Given the partially antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy, but also a certain elegance of design. This may be a thinly disguised version of the "salvation through suffering" philosophy, but in our case it worked.

Third: nearly from the start, the system was able to, and did, maintain itself. This fact is more important than it might seem. If designers of a system are forced to use that system, they quickly become aware of its functional and superficial deficiencies and are strongly motivated to correct them before it is too late. Because all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others.

The aspects of UNIX discussed in this paper exhibit clearly at least the first two of these design considerations. The interface to the file system, for example, is extremely convenient from a programming standpoint. The lowest possible interface level is designed to eliminate distinctions between the various devices and files and between direct and sequential access. No large "access method" routines are required to insulate the programmer from the system calls; in fact, all user programs either call the system directly or use a small library program, less than a page long, that buffers a number of characters and reads or writes them all at once.

Another important aspect of programming convenience is that there are no "control blocks" with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program's address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space.

Given the requirement that all programs should be usable with any file or device as input or output, it is also desirable to push device-dependent considerations into the operating system itself. The only alternatives seem to be to load, with all programs, routines for dealing with each device, which is expensive in space, or to depend on some means of dynamically linking to the routine appropriate to each device when it is actually needed, which is expensive either in overhead or in hardware.

Likewise, the process-control scheme and the command interface have proved both convenient and efficient. Because the shell operates as an ordinary, swappable user program, it consumes no "wired-down" space in the system proper, and it may be made as powerful as desired at little cost. In particular, given the framework in which the shell executes as a process that spawns other processes to perform commands, the notions of I/O redirection, background processes, command files, and user-selectable system interfaces all become essentially trivial to implement.

Influences

The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.

The **fork** operation, essentially as we implemented it, was present in the GENIE time-sharing system.[10] On a number of points we were influenced by Multics, which suggested the particular form of the I/O system calls[11] and both the name of the shell and its general functions. The notion that the shell should create a process for each command was also suggested to us by the early design of Multics, although in that system it was later dropped for efficiency reasons. A similar scheme is used by TENEX.[12]

IX. STATISTICS

The following numbers are presented to suggest the scale of the Research UNIX operation. Those of our users not involved in document preparation tend to use the system for program development, especially language work. There are few important "applications" programs.

Overall, we have today:

125	user population
33	maximum simultaneous users
1,630	directories
28,300	files
301,700	512-byte secondary storage blocks used

There is a "background" process that runs at the lowest possible priority; it is used to soak up any idle CPU time. It has been used to produce a million-digit approximation to the constant *e*, and other semi-infinite problems. Not counting this background work, we average daily:

13,500	commands
9.6	CPU hours
230	connect hours
62	different users
240	log-ins

X. ACKNOWLEDGMENTS

The contributors to UNIX are, in the traditional but here especially apposite phrase, too numerous to mention. Certainly, collective salutes are due to our colleagues in the Computing Science Research Center. R. H. Canaday contributed much to the basic design of the file system. We are particularly appreciative of the inventiveness, thoughtful criticism, and constant support of R. Morris, M. D. McIlroy, and J. F. Ossanna.

References

1. L. P. Deutsch and B. W. Lampson, "An online editor," *Comm. Assoc. Comp. Mach.* **10**(12) pp. 793-799, 803 (December 1967).
2. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.* **18** pp. 151-157 (March 1975).
3. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," *Bell Sys. Tech. J.* **57**(6) pp. 2115-2135 (1978).
4. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," *Proc. 2nd Int. Conf. on Software Engineering,* pp. 164-168 (October 13-15, 1976).
5. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," *Bell Sys. Tech. J.* **57**(6) pp. 2177-2200 (1978).

6. H. Lycklama, "UNIX Time-Sharing System: UNIX on a Microprocessor," *Bell Sys. Tech. J.* 57(6) pp. 2087-2101 (1978).

7. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).

8. Aleph-null, "Computer Recreations," *Software Practice and Experience* 1(2) pp. 201-204 (April-June 1971).

9. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," *Bell Sys. Tech. J.* 57(6) pp. 1971-1990 (1978).

10. L. P. Deutsch and B. W. Lampson, "SDS 930 time-sharing system preliminary reference manual," Doc. 30.10.10, Project GENIE, Univ. Cal. at Berkeley (April 1965).

11. R. J. Feiertag and E. I. Organick, "The Multics input-output system," *Proc. Third Symposium on Operating Systems Principles,* pp. 35-41 (October 18-20, 1971).

12. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Comm. Assoc. Comp. Mach.* 15(3) pp. 135-143 (March 1972).

UNIX For Beginners — Second Edition

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT .

This paper is meant to help new users get started on the UNIX† operating system. It includes:

- basics needed for day-to-day use of the system — typing commands, correcting typing mistakes, logging in and out, mail, inter-terminal communication, the file system, printing files, redirecting I/O, pipes, and the shell.
- document preparation — a brief discussion of the major formatting programs and macro packages, hints on preparing documents, and capsule descriptions of some supporting software.
- UNIX programming — using the editor, programming the shell, programming in C, other languages and tools.
- An annotated UNIX bibliography.

September 30, 1978

†UNIX is a Trademark of Bell Laboratories.

UNIX For Beginners — Second Edition

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

INTRODUCTION

From the user's point of view, the UNIX operating system is easy to learn and use, and presents few of the usual impediments to getting the job done. It is hard, however, for the beginner to know where to start, and how to make the best use of the facilities available. The purpose of this introduction is to help new users get used to the main ideas of the UNIX system and start making effective use of it quickly.

You should have a couple of other documents with you for easy reference as you read this one. The most important is *The UNIX Programmer's Manual*; it's often easier to tell you to read about something in the manual than to repeat its contents here. The other useful document is *A Tutorial Introduction to the UNIX Text Editor*, which will tell you how to use the editor to get text — programs, data, documents — into the computer.

A word of warning: the UNIX system has become quite popular, and there are several major variants in widespread use. Of course details also change with time. So although the basic structure of UNIX and how to use it is common to all versions, there will certainly be a few things which are different on your system from what is described here. We have tried to minimize the problem, but be aware of it. In cases of doubt, this paper describes Version 7 UNIX.

This paper has five sections:

1. Getting Started: How to log in, how to type, what to do about mistakes in typing, how to log out. Some of this is dependent on which system you log into (phone numbers, for example) and what terminal you use, so this section must necessarily be supplemented by local information.
2. Day-to-day Use: Things you need every day to use the system effectively: generally useful commands; the file system.
3. Document Preparation: Preparing manuscripts is one of the most common uses for UNIX systems. This section contains advice, but not extensive instructions on any of the formatting tools.
4. Writing Programs: UNIX is an excellent system for developing programs. This section talks about some of the tools, but again is not a tutorial in any of the programming languages provided by the system.
5. A UNIX Reading List. An annotated bibliography of documents that new users should be aware of.

I. GETTING STARTED

Logging In

You must have a UNIX login name, which you can get from whoever administers your system. You also need to know the phone number, unless your system uses permanently connected terminals. The UNIX system is capable of dealing with a wide variety of terminals: Terminet 300's; Execuport, TI and similar portables; video (CRT) terminals like the HP2640, etc.; high-priced graphics terminals like the Tektronix 4014; plotting terminals like those from GSI and DASI; and even the venerable Teletype in its various forms. But note: UNIX is strongly oriented towards devices with *lower case*. If your terminal produces only upper case (e.g., model 33 Teletype, some video and portable terminals), life will be so difficult that you should look for another terminal.

Be sure to set the switches appropriately on your device. Switches that might need to be adjusted include the speed, upper/lower case mode, full duplex, even parity, and any others that local wisdom advises. Establish a connection using whatever magic is needed for your terminal; this may involve dialing a telephone call or merely flipping a switch. In either case, UNIX should type "**login:**" at you. If it types garbage, you may be at the wrong speed; check the switches. If that fails, push the "break" or "interrupt" key a few times, slowly. If that fails to produce a login message, consult a guru.

When you get a **login:** message, type your login name *in lower case.* Follow it by a RETURN; the system will not do anything until you type a RETURN. If a password is required, you will be asked for it, and (if possible) printing will be turned off while you type it. Don't forget RETURN.

The culmination of your login efforts is a "prompt character," a single character that indicates that the system is ready to accept commands from you. The prompt character is usually a dollar sign $ or a percent sign %. (You may also get a message of the day just before the prompt character, or a notification that you have mail.)

Typing Commands

Once you've seen the prompt character, you can type commands, which are requests that the system do something. Try typing

```
date
```

followed by RETURN. You should get back something like

```
Mon Jan 16 14:17:10 EST 1978
```

Don't forget the RETURN after the command, or nothing will happen. If you think you're being ignored, type a RETURN; something should happen. RETURN won't be mentioned again, but don't forget it — it has to be there at the end of each line.

Another command you might try is **who**, which tells you everyone who is currently logged in:

```
who
```

gives something like

```
mb     tty01    Jan 16    09:11
ski    tty05    Jan 16    09:33
gam    tty11    Jan 16    13:07
```

The time is when the user logged in; "ttyxx" is the system's idea of what terminal the user is on.

If you make a mistake typing the command name, and refer to a non-existent command, you will be told. For example, if you type

```
whom
```

you will be told

```
whom: not found
```

Of course, if you inadvertently type the name of some other command, it will run, with more or less mysterious results.

Strange Terminal Behavior

Sometimes you can get into a state where your terminal acts strangely. For example, each letter may be typed twice, or the RETURN may not cause a line feed or a return to the left margin. You can often fix this by logging out and logging back in. Or you can read the description of the command stty in section I of the manual. To get intelligent treatment of tab characters (which are much used in UNIX) if your terminal doesn't have tabs, type the command

```
stty −tabs
```

and the system will convert each tab into the right number of blanks for you. If your terminal does have computer-settable tabs, the command **tabs** will set the stops correctly for you.

Mistakes in Typing

If you make a typing mistake, and see it before RETURN has been typed, there are two ways to recover. The sharp-character # erases the last character typed; in fact successive uses of # erase characters back to the beginning of the line (but not beyond). So if you type badly, you can correct as you go:

```
dd#atte##e
```

is the same as **date**.

The at-sign @ erases all of the characters typed so far on the current input line, so if the line is irretrievably fouled up, type an @ and start the line over.

What if you must enter a sharp or at-sign as part of the text? If you precede either # or @ by a backslash \, it loses its erase meaning. So to enter a sharp or at-sign in something, type \# or \@. The system will always echo a newline at you after your at-sign, even if preceded by a backslash. Don't worry — the at-sign has been recorded.

To erase a backslash, you have to type two sharps or two at-signs, as in \##. The backslash is used extensively in UNIX to indicate that the following character is in some way special.

Read-ahead

UNIX has full read-ahead, which means that you can type as fast as you want, whenever you want, even when some command is typing at you. If you type during output, your input characters will appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. So you can type several commands one after another without waiting for the first to finish or even begin.

Stopping a Program

You can stop most programs by typing the character "DEL" (perhaps called "delete" or "rubout" on your terminal). The "interrupt" or "break" key found on most terminals can also be used. In a few programs, like the text editor, DEL stops whatever the program is doing but leaves you in that program. Hanging up the phone will stop most programs.

Logging Out

The easiest way to log out is to hang up the phone. You can also type

login

and let someone else use the terminal you were on. It is usually not sufficient just to turn off the terminal. Most UNIX systems do not use a time-out mechanism, so you'll be there forever unless you hang up.

Mail

When you log in, you may sometimes get the message

You have mail.

UNIX provides a postal system so you can communicate with other users of the system. To read your mail, type the command

mail

Your mail will be printed, one message at a time, most recent message first. After each message, **mail** waits for you to say what to do with it. The two basic responses are **d**, which deletes the message, and RETURN, which does not (so it will still be there the next time you read your mailbox). Other responses are described in the manual. (Earlier versions of **mail** do not process one message at a time, but are otherwise similar.)

How do you send mail to someone else? Suppose it is to go to "joe" (assuming "joe" is someone's login name). The easiest way is this:

> **mail joe**
> *now type in the text of the letter*
> *on as many lines as you like ...*
> *After the last line of the letter*
> *type the character "control−d",*
> *that is, hold down "control" and type*
> *a letter "d".*

And that's it. The "control-d" sequence, often called "EOF" for end-of-file, is used throughout the system to mark the end of input from a terminal, so you might as well get used to it.

For practice, send mail to yourself. (This isn't as strange as it might sound — mail to oneself is a handy reminder mechanism.)

There are other ways to send mail — you can send a previously prepared letter, and you can mail to a number of people all at once. For more details see **mail**(1). (The notation **mail**(1) means the command **mail** in section 1 of the *UNIX Programmer's Manual.*)

Writing to other users

At some point, out of the blue will come a message like

Message from joe tty07...

accompanied by a startling beep. It means that Joe wants to talk to you, but unless you take explicit action you won't be able to talk back. To respond, type the command

write joe

This establishes a two-way communication path. Now whatever Joe types on his terminal will appear on yours and vice versa. The path is slow, rather like talking to the moon. (If you are in the middle of something, you have to get to a state where you can type a command. Normally, whatever program you are running has to terminate or be terminated. If you're editing, you can escape temporarily from the editor — read the editor tutorial.)

A protocol is needed to keep what you type from getting garbled up with what Joe types. Typically it's like this:

> Joe types **write smith** and waits.
> Smith types **write joe** and waits.
> Joe now types his message (as many lines as he likes). When he's ready for a reply, he signals it by typing **(o)**, which stands for "over".
> Now Smith types a reply, also terminated by **(o)**.
> This cycle repeats until someone gets tired; he then signals his intent to quit with **(oo)**, for "over and out".
> To terminate the conversation, each side must type a "control-d" character alone on a line. ("Delete" also works.) When the other person types his "control-d", you will get the message EOF on your terminal.

If you write to someone who isn't logged in, or who doesn't want to be disturbed, you'll be told. If the target is logged in but doesn't answer after a decent interval, simply type "control-d".

On-line Manual

The *UNIX Programmer's Manual* is typically kept on-line. If you get stuck on something, and can't find an expert to assist you, you can print on your terminal some manual section that might help. This is also useful for getting the most up-to-date information on a command. To print a manual section, type "man command-name". Thus to read up on the **who** command, type

```
man who
```

and, of course,

```
man man
```

tells all about the **man** command.

Computer Aided Instruction

Your UNIX system may have available a program called **learn**, which provides computer aided instruction on the file system and basic commands, the editor, docum nt preparation, and even C programming. Try typing the command

```
learn
```

If **learn** exists on your system, it will tell you what to do from there.

II. DAY-TO-DAY USE

Creating Files — The Editor

If you have to type a paper or a letter or a program, how do you get the information stored in the machine? Most of these tasks are done with the UNIX "text editor" **ed**. Since **ed** is thoroughly documented in **ed**(1) and explained in *A Tutorial Introduction to the UNIX Text Editor*, we won't spend any time here describing how to use it. All we want it for right now is to make some *files*. (A file is just a collection of information stored in the machine, a simplistic but adequate definition.)

To create a file called **junk** with some text in it, do the following:

```
ed junk     (invokes the text editor)
a           (command to "ed", to add text)
now type in
whatever text you want ...
.           (signals the end of adding text)
```

The "." that signals the end of adding text must be at the beginning of a line by itself. Don't forget it, for until it is typed, no other **ed** commands will be recognized — everything you type will be treated as text to be added.

At this point you can do various editing operations on the text you typed in, such as correcting spelling mistakes, rearranging paragraphs and the like. Finally, you must write the information you have typed into a file with the editor command **w**:

```
w
```

ed will respond with the number of characters it wrote into the file **junk**.

Until the **w** command, nothing is stored permanently, so if you hang up and go home the information is lost.† But after **w** the information is there permanently; you can re-access it any time by typing

```
ed junk
```

Type a **q** command to quit the editor. (If you try to quit without writing, **ed** will print a ? to remind you. A second **q** gets you out regardless.)

Now create a second file called **temp** in the same manner. You should now have two files, **junk** and **temp**.

What files are out there?

The **ls** (for "list") command lists the names (not contents) of any of the files that UNIX knows about. If you type

```
ls
```

the response will be

```
junk
temp
```

which are indeed the two files just created. The names are sorted into alphabetical order automatically, but other variations are possible. For example, the command

```
ls -t
```

causes the files to be listed in the order in which they were last changed, most recent first. The **−l** option gives a "long" listing:

```
ls -l
```

will produce something like

```
-rw-rw-rw-  1 bwk   41 Jul 22 2:56 junk
-rw-rw-rw-  1 bwk   78 Jul 22 2:57 temp
```

The date and time are of the last change to the file. The 41 and 78 are the number of characters (which should agree with the numbers you got from **ed**). **bwk** is the owner of the file, that is, the person who created it. The **−rw−rw−rw−** tells who has permission to read and write the file, in this case everyone.

† This is not strictly true — if you hang up while editing, the data you were working on is saved in a file called **ed.hup**, which you can continue with at your next session.

Options can be combined: **ls −lt** gives the same thing as **ls −l**, but sorted into time order. You can also name the files you're interested in, and **ls** will list the information about them only. More details can be found in ls(1).

The use of optional arguments that begin with a minus sign, like **−t** and **−lt**, is a common convention for UNIX programs. In general, if a program accepts such optional arguments, they precede any filename arguments. It is also vital that you separate the various arguments with spaces: **ls−l** is not the same as **ls −l**.

Printing Files

Now that you've got a file of text, how do you print it so people can look at it? There are a host of programs that do that, probably more than are needed.

One simple thing is to use the editor, since printing is often done just before making changes anyway. You can say

```
ed junk
1,$p
```

ed will reply with the count of the characters in **junk** and then print all the lines in the file. After you learn how to use the editor, you can be selective about the parts you print.

There are times when it's not feasible to use the editor for printing. For example, there is a limit on how big a file **ed** can handle (several thousand lines). Secondly, it will only print one file at a time, and sometimes you want to print several, one after another. So here are a couple of alternatives.

First is **cat**, the simplest of all the printing programs. **cat** simply prints on the terminal the contents of all the files named in a list. Thus

```
cat junk
```

prints one file, and

```
cat junk temp
```

prints two. The files are simply concatenated (hence the name "**cat**") onto the terminal.

pr produces formatted printouts of files. As with **cat**, **pr** prints all the files named in a list. The difference is that it produces headings with date, time, page number and file name at the top of each page, and extra lines to skip over the fold in the paper. Thus,

```
pr junk temp
```

will print **junk** neatly, then skip to the top of a new page and print **temp** neatly.

pr can also produce multi-column output:

```
pr −3 junk
```

prints **junk** in 3-column format. You can use any reasonable number in place of "3" and **pr** will do its best. **pr** has other capabilities as well: see **pr**(1).

It should be noted that **pr** is *not* a formatting program in the sense of shuffling lines around and justifying margins. The true formatters are **nroff** and **troff**, which we will get to in the section on document preparation.

There are also programs that print files on a high-speed printer. Look in your manual under **opr** and **lpr**. Which to use depends on what equipment is attached to your machine.

Shuffling Files About

Now that you have some files in the file system and some experience in printing them, you can try bigger things. For example, you can move a file from one place to another (which amounts to giving it a new name), like this:

```
mv junk precious
```

This means that what used to be "junk" is now "precious". If you do an **ls** command now, you will get

```
precious
temp
```

Beware that if you move a file to another one that already exists, the already existing contents are lost forever.

If you want to make a *copy* of a file (that is, to have two versions of something), you can use the **cp** command:

```
cp precious temp1
```

makes a duplicate copy of **precious** in **temp1**.

Finally, when you get tired of creating and moving files, there is a command to remove files from the file system, called **rm**.

```
rm temp temp1
```

will remove both of the files named.

You will get a warning message if one of the named files wasn't there, but otherwise **rm**, like most UNIX commands, does its work silently. There is no prompting or chatter, and error messages are occasionally curt. This terseness is sometimes disconcerting to newcomers, but experienced users find it desirable.

What's in a Filename

So far we have used filenames without ever saying what's a legal name, so it's time for a couple of rules. First, filenames are limited to 14 characters, which is enough to be descriptive.

Second, although you can use almost any character in a filename, common sense says you should stick to ones that are visible, and that you should probably avoid characters that might be used with other meanings. We have already seen, for example, that in the **ls** command, **ls −t** means to list in time order. So if you had a file whose name was **−t**, you would have a tough time listing it by name. Besides the minus sign, there are other characters which have special meaning. To avoid pitfalls, you would do well to use only letters, numbers and the period until you're familiar with the situation.

On to some more positive suggestions. Suppose you're typing a large document like a book. Logically this divides into many small pieces, like chapters and perhaps sections. Physically it must be divided too, for **ed** will not handle really big files. Thus you should type the document as a number of files. You might have a separate file for each chapter, called

```
chap1
chap2
etc...
```

Or, if each chapter were broken into several files, you might have

```
chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...
```

You can now tell at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice UNIX user. What if you wanted to print the whole book? You could say

```
pr chap1.1 chap1.2 chap1.3 ......
```

but you would get tired pretty fast, and would probably even make mistakes. Fortunately, there is a shortcut. You can say

```
pr chap*
```

The * means "anything at all," so this translates into "print all files whose names begin with **chap**", listed in alphabetical order.

This shorthand notation is not a property of the **pr** command, by the way. It is system-wide, a service of the program that interprets commands (the "shell," sh(1)). Using that fact, you can see how to list the names of the files in the book:

```
ls chap*
```

produces

```
chap1.1
chap1.2
chap1.3
...
```

The * is not limited to the last position in a filename — it can be anywhere and can occur several times. Thus

```
rm *junk* *temp*
```

removes all files that contain **junk** or **temp** as any part of their name. As a special case, * by itself matches every filename, so

```
pr *
```

prints all your files (alphabetical order), and

```
rm *
```

removes *all files.* (You had better be *very* sure that's what you wanted to say!)

The * is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4 and 9. Then you can say

```
pr chap[12349]*
```

The [...] means to match any of the characters inside the brackets. A range of consecutive letters or digits can be abbreviated, so you can also do this with

```
pr chap[1−49]*
```

Letters can also be used within brackets: **[a−z]** matches any character in the range **a** through **z**.

The ? pattern matches any single character, so

```
ls ?
```

lists all files which have single-character names, and

```
ls −l chap?.1
```

lists information about the first file of each chapter (**chap1.1**, **chap2.1**, etc.).

Of these niceties, * is certainly the most useful, and you should get used to it. The others are frills, but worth knowing.

If you should ever have to turn off the special meaning of *, ?, etc., enclose the entire argument in single quotes, as in

```
ls '?'
```

We'll see some more examples of this shortly.

What's in a Filename, Continued

When you first made that file called **junk**, how did the system know that there wasn't another **junk** somewhere else, especially since the person in the next office is also reading this tutorial? The answer is that generally each user has a private *directory*, which contains only the files that belong to him. When you log in, you are "in" your directory. Unless you take special action, when you create a new file, it is made in the directory that you are currently in; this is most often your own directory, and thus the file is unrelated to any other file of the same name that might exist in someone else's directory.

The set of all files is organized into a (usually big) tree, with your files located several branches into the tree. It is possible for you to "walk" around this tree, and to find any file in the system, by starting at the root of the tree and walking along the proper set of branches. Conversely, you can start where you are and walk toward the root.

Let's try the latter first. The basic tools is the command **pwd** ("print working directory"), which prints the name of the directory you are currently in.

Although the details will vary according to the system you are on, if you give the command **pwd**, it will print something like

```
/usr/your-name
```

This says that you are currently in the directory **your-name**, which is in turn in the directory /usr, which is in turn in the root directory called by convention just /. (Even if it's not called /usr on your system, you will get something analogous. Make the corresponding changes and read on.)

If you now type

```
ls /usr/your-name
```

you should get exactly the same list of file names as you get from a plain ls: with no arguments, ls lists the contents of the current directory; given the name of a directory, it lists the contents of that directory.

Next, try

```
ls /usr
```

This should print a long series of names, among which is your own login name **your-name**. On many systems, usr is a directory that contains the directories of all the normal users of the system, like you.

The next step is to try

```
ls /
```

You should get a response something like this (although again the details may be different):

```
bin
dev
etc
lib
tmp
usr
```

This is a collection of the basic directories of files that the system knows about; we are at the root of the tree.

Now try

```
cat /usr/your-name/junk
```

(if **junk** is still around in your directory). The name

```
/usr/your-name/junk
```

is called the **pathname** of the file that you normally think of as "junk". "Pathname" has an obvious meaning: it represents the full name of the path you have to follow from the root through the tree of directories to get to a particular file. It is a universal rule in the UNIX system that anywhere you can use an ordinary filename, you can use a pathname.

Here is a picture which may make this clearer:

```
                    (root)
                     /|\
                    / | \
                   /  |  \
        bin    etc    usr    dev   tmp
        /|\    /|\    /|\    /|\   /|\
                     / | \
                    /  |  \
               adam   eve   mary
               /     / \      \
                    /   \     junk
                junk temp
```

Notice that Mary's **junk** is unrelated to Eve's.

This isn't too exciting if all the files of interest are in your own directory, but if you work with someone else or on several projects concurrently, it becomes handy indeed. For example, your friends can print your book by saying

```
pr /usr/your-name/chap*
```

Similarly, you can find out what files your neighbor has by saying

```
ls /usr/neighbor-name
```

or make your own copy of one of his files by

```
cp /usr/your-neighbor/his-file yourfile
```

If your neighbor doesn't want you poking around in his files, or vice versa, privacy can be arranged. Each file and directory has read-write-execute permissions for the owner, a group, and everyone else, which can be set to control access. See **ls**(1) and **chmod**(1) for details. As a matter of observed fact, most users most of the time find openness of more benefit than privacy.

As a final experiment with pathnames, try

```
ls /bin /usr/bin
```

Do some of the names look familiar? When you run a program, by typing its name after the prompt character, the system simply looks for a file of that name. It normally looks first in your directory (where it typically doesn't find it), then in **/bin** and finally in **/usr/bin**. There is nothing magic about commands like **cat** or **ls**, except that they have been collected into a couple of places to be easy to find and administer.

What if you work regularly with someone else on common information in his directory? You could just log in as your friend each time you want to, but you can also say "I want to work on his files instead of my own". This is done by changing the directory that you are currently in:

```
cd /usr/your-friend
```

(On some systems, **cd** is spelled **chdir**.) Now when you use a filename in something like **cat** or **pr**, it refers to the file in your friend's directory. Changing directories doesn't affect any permissions associated with a file — if you couldn't access a file from your own directory, changing to another directory won't alter that fact. Of course, if you forget what directory you're in, type

```
pwd
```

to find out.

It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, when you write your book, you might want to keep all the text in a directory called **book**. So make one with

```
mkdir book
```

then go to it with

```
cd book
```

then start typing chapters. The book is now found in (presumably)

```
/usr/your-name/book
```

To remove the directory **book**, type

```
rm book/*
rmdir book
```

The first command removes all files from the directory; the second removes the empty directory.

You can go up one level in the tree of files by saying

```
cd ..
```

".." is the name of the parent of whatever directory you are currently in. For completeness, "." is an alternate name for the directory you are in.

Using Files instead of the Terminal

Most of the commands we have seen so far produce output on the terminal; some, like the editor, also take their input from the terminal. It is universal in UNIX systems that the terminal can be replaced by a file for either or both of input and output. As one example,

```
ls
```

makes a list of files on your terminal. But if you say

```
ls >filelist
```

a list of your files will be placed in the file **filelist** (which will be created if it doesn't already exist, or overwritten if it does). The symbol > means "put the output on the following file, rather than on the terminal." Nothing is produced on the terminal. As another example, you could combine several files into one by capturing the output of **cat** in a file:

```
cat f1 f2 f3 >temp
```

The symbol >> operates very much like > does, except that it means "add to the end of." That is,

```
cat f1 f2 f3 >>temp
```

means to concatenate **f1**, **f2** and **f3** to the end of whatever is already in **temp**, instead of overwriting the existing contents. As with >, if **temp** doesn't exist, it will be created for you.

In a similar way, the symbol < means to take the input for a program from the following file, instead of from the terminal. Thus, you could make up a script of commonly used editing commands and put them into a file called **script**. Then you can run the script on a file by saying

```
ed file <script
```

As another example, you can use **ed** to prepare a letter in file **let**, then send it to several people with

```
mail adam eve mary joe <let
```

Pipes

One of the novel contributions of the UNIX system is the idea of a *pipe*. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes — a pipeline.

For example,

```
pr f g h
```

will print the files f, g, and h, beginning each on a new page. Suppose you want them run together instead. You could say

```
cat f g h >temp
pr <temp
rm temp
```

but this is more work than necessary. Clearly what we want is to take the output of **cat** and connect it to the input of **pr**. So let us use a pipe:

```
cat f g h | pr
```

The vertical bar | means to take the output from **cat**, which would normally have gone to the terminal, and put it into **pr** to be neatly formatted.

There are many other examples of pipes. For example,

```
ls | pr -3
```

prints a list of your files in three columns. The program **wc** counts the number of lines, words and characters in its input, and as we saw earlier, **who** prints a list of currently-logged on people, one per line. Thus

```
who | wc
```

tells how many people are logged on. And of course

```
ls | wc
```

counts your files.

Any program that reads from the terminal can read from a pipe instead; any program that writes on the terminal can drive a pipe. You can have as many elements in a pipeline as you wish.

Many UNIX programs are written so that they will take their input from one or more files if file arguments are given; if no arguments are given they will read from the terminal, and thus can be used in pipelines. **pr** is one example:

```
pr -3 a b c
```

prints files a, b and c in order in three columns. But in

```
cat a b c | pr -3
```

pr prints the information coming down the pipeline, still in three columns.

The Shell

We have already mentioned once or twice the mysterious "shell," which is in fact **sh**(1). The shell is the program that interprets what you type as commands and arguments. It also looks after translating *, etc., into lists of filenames, and <, >, and | into changes of input and output streams.

The shell has other capabilities too. For example, you can run two programs with one command line by separating the commands with a semicolon; the shell recognizes the semicolon and breaks the line into two commands. Thus

```
date; who
```

does both commands before returning with a prompt character.

You can also have more than one program running *simultaneously* if you wish. For example, if you are doing something time-consuming, like the editor script of an earlier section, and you don't want to wait around for the results before starting something else, you can say

```
ed file <script &
```

The ampersand at the end of a command line says "start this command running, then take further commands from the terminal immediately," that is, don't wait for it to complete. Thus the script will begin, but you can do something else at the same time. Of course, to keep the output from interfering with what you're doing on the terminal, it would be better to say

```
ed file <script >script.out &
```

which saves the output lines in a file called **script.out**.

When you initiate a command with &, the system replies with a number called the process number, which identifies the command in case you later want to stop it. If you do, you can say

```
kill process-number
```

If you forget the process number, the command **ps** will tell you about everything you have running. (If you are desperate, **kill 0** will kill all your processes.) And if you're curious about other people, **ps a** will tell you about *all* programs that are currently running.

You can say

```
(command-1; command-2; command-3) &
```

to start three commands in the background, or you can start a background pipeline with

```
command-1 | command-2 &
```

Just as you can tell the editor or some similar program to take its input from a file instead of from the terminal, you can tell the shell to read a file to get commands. (Why not? The shell, after all, is just a program, albeit a clever one.) For instance, suppose you want to set tabs on your terminal, and find out the date and who's on the system every time you log in. Then you can put the three necessary commands (**tabs**, **date**, **who**) into a file, let's call it **startup**, and then run it with

```
sh startup
```

This says to run the shell with the file **startup** as input. The effect is as if you had typed the contents of **startup** on the terminal.

If this is to be a regular thing, you can eliminate the need to type **sh**: simply type, once only, the command

```
chmod +x startup
```

and thereafter you need only say

```
startup
```

to run the sequence of commands. The **chmod**(1) command marks the file executable; the shell recognizes this and runs it as a sequence of commands.

If you want **startup** to run automatically every time you log in, create a file in your login directory called **.profile**, and place in it the line **startup**. When the shell first gains control when you log in, it looks for the **.profile** file and does whatever commands it finds in it. We'll get back to the shell in the section on programming.

III. DOCUMENT PREPARATION

UNIX systems are used extensively for document preparation. There are two major formatting programs, that is, programs that produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, and the like. **nroff** is designed to produce output on terminals and line-printers. **troff** (pronounced "tee-roff") instead drives a phototypesetter, which produces very high quality output on photographic paper. This paper was formatted with **troff**.

Formatting Packages

The basic idea of **nroff** and **troff** is that the text to be formatted contains within it "formatting commands" that indicate in detail how the formatted text is to look. For example, there might be commands that specify how long lines are, whether to use single or double spacing, and what running titles to use on each page.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several "packages" of canned formatting requests are available to let you specify paragraphs, running titles, footnotes, multi-column output, and so on, with little effort and without having to learn **nroff** and **troff**. These packages take a modest effort to learn, but the rewards for using them are so great that it is time well spent.

In this section, we will provide a hasty look at the "manuscript" package known as **−ms**. Formatting requests typically consist of a period and two upper-case letters, such as **.TL**, which is used to introduce a title, or **.PP** to begin a new paragraph.

A document is typed so it looks something like this:

```
.TL
title of document
.AU
author name
.SH
section heading
.PP
paragraph ...
.PP
another paragraph ...
.SH
another section heading
.PP
etc.
```

The lines that begin with a period are the formatting requests. For example, **.PP** calls for starting a new paragraph. The precise meaning of **.PP** depends on what output device is being used (typesetter or terminal, for instance), and on what publication the document will appear in. For example, **−ms** normally assumes that a paragraph is preceded by a space (one line in **nroff**, ½ line in **troff**), and the first word is indented. These rules can be changed if you like, but they are changed by changing the interpretation of **.PP**, not by re-typing the document.

To actually produce a document in standard format using **−ms**, use the command

```
troff −ms files ...
```

for the typesetter, and

```
nroff −ms files ...
```

for a terminal. The **−ms** argument tells **troff** and **nroff** to use the manuscript package of formatting requests.

There are several similar packages; check with a local expert to determine which ones are in common use on your machine.

Supporting Tools

In addition to the basic formatters, there is a host of supporting programs that help with document preparation. The list in the next few paragraphs is far from complete, so browse through the manual and check with people around you for other possibilities.

eqn and **neqn** let you integrate mathematics into the text of a document, in an easy-to-learn language that closely resembles the way you would speak it aloud. For example, the **eqn** input

```
sum from i=0 to n x sub i ~=~ pi over 2
```

produces the output

$$\sum_{i=0}^{n} x_i ~=~ \frac{\pi}{2}$$

The program **tbl** provides an analogous service for preparing tabular material; it does all the computations necessary to align complicated columns with elements of varying widths.

refer prepares bibliographic citations from a data base, in whatever style is defined by the formatting package. It looks after all the details of numbering references in sequence, filling in page and volume numbers, getting the author's initials and the journal name right, and so on.

spell and **typo** detect possible spelling mistakes in a document. **spell** works by comparing the words in your document to a dictionary, printing those that are not in the dictionary. It knows enough about English spelling to detect plurals and the like, so it does a very good job. **typo** looks for words which are "unusual", and prints those. Spelling mistakes tend to be more unusual, and thus show up early when the most unusual words are printed first.

grep looks through a set of files for lines that contain a particular text pattern (rather like the editor's context search does, but on a bunch of files). For example,

```
grep 'ing$' chap*
```

will find all lines that end with the letters **ing** in the files **chap***. (It is almost always a good practice to put single quotes around the pattern you're searching for, in case it contains characters like * or $ that have a special meaning to the shell.) **grep** is often useful for finding out in which of a set of files the misspelled words detected by **spell** are actually located.

diff prints a list of the differences between two files, so you can compare two versions of something automatically (which certainly beats proofreading by hand).

wc counts the words, lines and characters in a set of files. **tr** translates characters into other characters; for example it will convert upper to lower case and vice versa. This translates upper into lower:

```
tr A-Z a-z <input >output
```

sort sorts files in a variety of ways: **cref** makes cross-references; **ptx** makes a permuted index (keyword-in-context listing). **sed** provides many of the editing facilities of **ed**, but can apply them to arbitrarily long inputs. **awk** provides the ability to do both pattern matching and numeric computations, and to conveniently process fields within lines. These programs are for more advanced users, and they are not limited to document preparation. Put them on your list of things to learn about.

Most of these programs are either independently documented (like **eqn** and **tbl**), or are sufficiently simple that the description in the *UNIX Programmer's Manual* is adequate explanation.

Hints for Preparing Documents

Most documents go through several versions (always more than you expected) before they are finally finished. Accordingly, you should do whatever possible to make the job of changing them easy.

First, when you do the purely mechanical operations of typing, type so that subsequent editing will be easy. Start each sentence on a new line. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting and rearranging sentences, these precautions simplify any editing you have to do later.

Keep the individual files of a document down to modest size, perhaps ten to fifteen thousand characters. Larger files edit more slowly, and of course if you make a dumb mistake it's better to have clobbered a small file than a big one. Split into files at natural boundaries in the document, for the same reasons that you start each sentence on a new line.

The second aspect of making change easy is to not commit yourself to formatting details too early. One of the advantages of formatting packages like **-ms** is that they permit you to delay decisions to the last possible moment. Indeed, until a document is printed, it is not even decided whether it will be typeset or put on a line printer.

As a rule of thumb, for all but the most trivial jobs, you should type a document in terms of a set of requests like .PP, and then define them appropriately, either by using one of the canned packages (the better way) or by defining your own **nroff** and **troff** commands. As long as you have entered the text in some systematic way, it can always be cleaned up and re-formatted by a judicious combination of editing commands and request definitions.

IV. PROGRAMMING

There will be no attempt made to teach any of the programming languages available but a few words of advice are in order. One of the reasons why the UNIX system is a productive programming environment is that there is already a rich set of tools available, and facilities like pipes, I/O redirection, and the capabilities of the shell often make it possible to do a job by pasting together programs that already exist instead of writing from scratch.

The Shell

The pipe mechanism lets you fabricate quite complicated operations out of spare parts that already exist. For example, the first draft of the **spell** program was (roughly)

```
cat ...     collect the files
| tr ...    put each word on a new line
| tr ...    delete punctuation, etc.
| sort      into dictionary order
| uniq      discard duplicates
| comm      print words in text
              but not in dictionary
```

More pieces have been added subsequently, but this goes a long way for such a small effort.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, such as a book, you could laboriously type

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
etc.
```

But you can do the job much more easily. One way is to type

```
ls chap* >temp
```

to get the list of filenames into a file. Then edit this file to make the necessary series of editing commands (using the global commands of **ed**), and write it into **script**. Now the command

```
ed <script
```

will produce the same output as the laborious hand typing. Alternately (and more easily), you can use the fact that the shell will perform loops, repeating a set of commands over and over again for a set of arguments:

```
for i in chap*
do
        ed $i <script
done
```

This sets the shell variable **i** to each file name in turn, then does the command. You can type this command at the terminal, or put it in a file for later execution.

Programming the Shell

An option often overlooked by newcomers is that the shell is itself a programming language, with variables, control flow (**if-else, while, for, case**), subroutines, and interrupt handling. Since there are many building-block programs, you can sometimes avoid writing a new program merely by piecing together some of the building blocks with shell command files.

We will not go into any details here; examples and rules can be found in *An Introduction to the UNIX Shell*, by S. R. Bourne.

Programming in C

If you are undertaking anything substantial, C is the only reasonable choice of programming language: everything in the UNIX system is tuned to it. The system itself is written in C, as are most of the programs that run on it. It is also a easy language to use once you get started. C is introduced and fully described in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Several sections of the manual describe the system interfaces, that is, how you do I/O and similar functions. Read *UNIX Programming* for more complicated things.

Most input and output in C is best handled with the standard I/O library, which provides a set of I/O functions that exist in compatible form on most machines that have C compilers. In general, it's wisest to confine the system interactions in a program to the facilities provided by this library.

C programs that don't depend too much on special features of UNIX (such as pipes) can be moved to other computers that have C compilers. The list of such machines grows daily; in addition to the original PDP-11, it currently includes at least Honeywell 6000, IBM 370, Interdata 8/32, Data General Nova and Eclipse, HP 2100, Harris /7, VAX 11/780, SEL 86, and Zilog Z80. Calls to the standard I/O library will work on all of these machines.

There are a number of supporting programs that go with C. **lint** checks C programs for potential portability problems, and detects errors such as mismatched argument types and uninitialized variables.

For larger programs (anything whose source is on more than one file) **make** allows you to specify the dependencies among the source files and the processing steps needed to make a new version; it then checks the times that the pieces were last changed and does the minimal amount of recompiling to create a consistent updated version.

The debugger **adb** is useful for digging through the dead bodies of C programs, but is rather hard to learn to use effectively. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

The C compiler provides a limited instrumentation service, so you can find out where programs spend their time and what parts are worth optimizing. Compile the routines with the **−p** option; after the test run, use **prof** to print an execution profile. The command **time** will give you the gross run-time statistics of a program, but they are not super accurate or reproducible.

Other Languages

If you *have* to use Fortran, there are two possibilities. You might consider Ratfor, which gives you the decent control structures and free-form input that characterize C, yet lets you write code that is still portable to other environments. Bear in mind that UNIX Fortran tends to produce large and relatively slow-running programs. Furthermore, supporting software like **adb**, **prof**, etc., are all virtually useless with Fortran programs. There may also be a Fortran 77 compiler on your system. If so, this is a viable alternative to Ratfor, and has the non-trivial advantage that it is compatible with C and related programs. (The Ratfor processor and C tools can be used with Fortran 77 too.)

If your application requires you to translate a language into a set of actions or another language, you are in effect building a compiler, though probably a small one. In that case, you should be using the **yacc** compiler-compiler, which helps you develop a compiler quickly. The **lex** lexical analyzer generator does the same job for the simpler languages that can be expressed as regular expressions. It can be used by itself, or as a front end to recognize inputs for a **yacc**-based program. Both **yacc** and **lex** require some sophistication to use, but the initial effort of learning them can be repaid many times over in programs that are easy to change later on.

Most UNIX systems also make available other languages, such as Algol 68, APL, Basic, Lisp, Pascal, and Snobol. Whether these are useful depends largely on the local environment: if someone cares about the language and has worked on it, it may be in good shape. If not, the odds are strong that it will be more trouble than it's worth.

V. UNIX READING LIST

General:

K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual*, Bell Laboratories, 1978. Lists commands, system routines and interfaces, file formats, and some of the maintenance procedures. You can't live without this, although you will probably only need to read section 1.

Documents for Use with the UNIX Time-sharing System. Volume 2 of the Programmer's Manual. This contains more extensive descriptions of major commands, and tutorials and reference manuals. All of the papers listed below are in it, as are descriptions of most of the programs mentioned above.

D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System," CACM, July 1974. An overview of the system, for people interested in operating systems. Worth reading by anyone who programs. Contains a remarkable number of one-sentence observations on how to do things right.

The Bell System Technical Journal (BSTJ) Special Issue on UNIX, July/August, 1978, contains many papers describing recent developments, and some retrospective material.

The 2nd International Conference on Software Engineering (October, 1976) contains several papers describing the use of the Programmer's Workbench (PWB) version of UNIX.

Document Preparation:

B. W. Kernighan, "A Tutorial Introduction to the UNIX Text Editor" and "Advanced Editing on UNIX," Bell Laboratories, 1978. Beginners need the introduction; the advanced material will help you get the most out of the editor.

M. E. Lesk, "Typing Documents on UNIX," Bell Laboratories, 1978. Describes the **−ms** macro package, which isolates the novice from the vagaries of **nroff** and **troff**, and takes care of most formatting situations. If this specific package isn't available on your system, something similar probably is. The most likely alternative is the PWB/UNIX macro package **−mm**; see your local guru if you use PWB/UNIX.

B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," Bell Laboratories Computing Science Tech. Rep. 17.

M. E. Lesk, "Tbl — A Program to Format Tables," Bell Laboratories CSTR 49, 1976.

J. F. Ossanna, Jr., "NROFF/TROFF User's Manual," Bell Laboratories CSTR 54, 1976. **troff** is the basic formatter used by **−ms**, **eqn** and **tbl**. The reference manual is indispensable if you are going to write or maintain these or similar programs. But start with:

B. W. Kernighan, "A TROFF Tutorial," Bell Laboratories, 1976. An attempt to unravel the intricacies of **troff**.

Programming:

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978. Contains a tutorial introduction, complete discussions of all language features, and the reference manual.

B. W. Kernighan and D. M. Ritchie, "UNIX Programming," Bell Laboratories, 1978. Describes how to interface with the system from C programs: I/O calls, signals, processes.

S. R. Bourne, "An Introduction to the UNIX Shell," Bell Laboratories, 1978. An introduction and reference manual for the Version 7 shell. Mandatory reading if you intend to make effective use of the programming power of this shell.

S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Bell Laboratories CSTR 32, 1978.

M. E. Lesk, "Lex — A Lexical Analyzer Generator," Bell Laboratories CSTR 39, 1975.

S. C. Johnson, "Lint, a C Program Checker," Bell Laboratories CSTR 65, 1977.

S. I. Feldman, "MAKE — A Program for Maintaining Computer Programs," Bell Laboratories CSTR 57, 1977.

J. F. Maranzano and S. R. Bourne, "A Tutorial Introduction to ADB," Bell Laboratories CSTR 62, 1977. An introduction to a powerful but complex debugging tool.

S. I. Feldman and P. J. Weinberger, "A Portable Fortran 77 Compiler," Bell Laboratories, 1978. A full Fortran 77 for UNIX systems.

LEARN — Computer-Aided Instruction on UNIX
(Second Edition)

Brian W. Kernighan

Michael E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the second version of the *learn* program for interpreting CAI scripts on the UNIX† operating system, and a set of scripts that provide a computerized introduction to the system.

Six current scripts cover basic commands and file handling, the editor, additional file handling commands, the *eqn* program for mathematical typing, the "−ms" package of formatting macros, and an introduction to the C programming language. These scripts now include a total of about 530 lessons.

Many users from a wide variety of backgrounds have used *learn* to acquire basic UNIX skills. Most usage involves the first two scripts, an introduction to files and commands, and the text editor.

The second version of *learn* is about four times faster than the previous one in CPU utilization, and much faster in perceived time because of better overlap of computing and printing. It also requires less file space than the first version. Many of the lessons have been revised; new material has been added to reflect changes and enhancements in the UNIX system itself. Script-writing is also easier because of revisions to the script language.

January 30, 1979

†UNIX is a Trademark of Bell Laboratories.

LEARN — Computer-Aided Instruction on UNIX (Second Edition)

Brian W. Kernighan

Michael E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction.

Learn is a driver for CAI scripts. It is intended to permit the easy composition of lessons and lesson fragments to teach people computer skills. Since it is teaching the same system on which it is implemented, it makes direct use of UNIX† facilities to create a controlled UNIX environment. The system includes two main parts: (1) a driver that interprets the lesson scripts; and (2) the lesson scripts themselves. At present there are six scripts:

- basic file handling commands
- the UNIX text editor *ed*
- advanced file handling
- the *eqn* language for typing mathematics
- the "−ms" macro package for document formatting
- the C programming language

The purported advantages of CAI scripts for training in computer skills include the following:

(a) students are forced to perform the exercises that are in fact the basis of training in any case;

(b) students receive immediate feedback and confirmation of progress;

(c) students may progress at their own rate;

(d) no schedule requirements are imposed; students may study at any time convenient for them;

(e) the lessons may be improved individually and the improvements are immediately available to new users;

(f) since the student has access to a computer for the CAI script there is a place to do exercises;

(g) the use of high technology will improve student motivation and the interest of their management.

Opposed to this, of course, is the absence of anyone to whom the student may direct questions. If CAI is used without a "counselor" or other assistance, it should properly be compared to a textbook, lecture series, or taped course, rather than to a seminar. CAI has been used for many years in a variety of educational areas.[1,2,3] The use of a computer to teach itself, however, offers unique advantages. The skills developed to get through the script are exactly those needed to use the computer; there is no waste effort.

The scripts written so far are based on some familiar assumptions about education; these

†UNIX is a Trademark of Bell Laboratories.

assumptions are outlined in the next section. The remaining sections describe the operation of the script driver and the particular scripts now available. The driver puts few restrictions on the script writer, but the current scripts are of a rather rigid and stereotyped form in accordance with the theory in the next section and practical limitations.

2. Educational Assumptions and Design.

First, the way to teach people how to do something is to have them do it. Scripts should not contain long pieces of explanation; they should instead frequently ask the student to do some task. So teaching is always by example: the typical script fragment shows a small example of some technique and then asks the user to either repeat that example or produce a variation on it. All are intended to be easy enough that most students will get most questions right, reinforcing the desired behavior.

Most lessons fall into one of three types. The simplest presents a lesson and asks for a yes or no answer to a question. The student is given a chance to experiment before replying. The script checks for the correct reply. Problems of this form are sparingly used.

The second type asks for a word or number as an answer. For example a lesson on files might say

> *How many files are there in the current directory? Type "answer N", where N is the number of files.*

The student is expected to respond (perhaps after experimenting) with

> *answer 17*

or whatever. Surprisingly often, however, the idea of a substitutable argument (i.e., replacing *N* by 17) is difficult for non-programmer students, so the first few such lessons need real care.

The third type of lesson is open-ended — a task is set for the student, appropriate parts of the input or output are monitored, and the student types *ready* when the task is done. Figure 1 shows a sample dialog that illustrates the last of these, using two lessons about the *cat* (concatenate, i.e., print) command taken from early in the script that teaches file handling. Most *learn* lessons are of this form.

After each correct response the computer congratulates the student and indicates the lesson number that has just been completed, permitting the student to restart the script after that lesson. If the answer is wrong, the student is offered a chance to repeat the lesson. The "speed" rating of the student (explained in section 5) is given after the lesson number when the lesson is completed successfully; it is printed only for the aid of script authors checking out possible errors in the lessons.

It is assumed that there is no foolproof way to determine if the student truly "understands" what he or she is doing; accordingly, the current *learn* scripts only measure performance, not comprehension. If the student can perform a given task, that is deemed to be "learning."[4]

The main point of using the computer is that what the student does is checked for correctness immediately. Unlike many CAI scripts, however, these scripts provide few facilities for dealing with wrong answers. In practice, if most of the answers are not right the script is a failure; the universal solution to student error is to provide a new, easier script. Anticipating possible wrong answers is an endless job, and it is really easier as well as better to provide a simpler script.

Along with this goes the assumption that anything can be taught to anybody if it can be broken into sufficiently small pieces. Anything not absorbed in a single chunk is just subdivided.

To avoid boring the faster students, however, an effort is made in the files and editor scripts to provide three tracks of different difficulty. The fastest sequence of lessons is aimed at roughly the bulk and speed of a typical tutorial manual and should be adequate for review and for well-prepared students. The next track is intended for most users and is roughly twice as

Figure 1: Sample dialog from basic files script

(Student responses in italics; '$' is the prompt)

```
A file can be printed on your terminal
by using the "cat" command. Just say
"cat file" where "file" is the file name.
For example, there is a file named
"food" in this directory. List it
by saying "cat food"; then type "ready".
$ cat food
  this is the file
  named food.
$ ready

Good. Lesson 3.3a (1)

Of course, you can print any file with "cat".
In particular, it is common to first use
"ls" to find the name of a file and then "cat"
to print it. Note the difference between
"ls", which tells you the name of the file,
and "cat", which tells you the contents.
One file in the current directory is named for
a President. Print the file, then type "ready".
$ cat President
cat: can't open President
$ ready

Sorry, that's not right. Do you want to try again? yes
Try the problem again.
$ ls
.ocopy
X1
roosevelt
$ cat roosevelt
  this file is named roosevelt
  and contains three lines of
  text.
$ ready

Good. Lesson 3.3b (0)

The "cat" command can also print several files
at once. In fact, it is named "cat" as an abbreviation
for "concatenate"....
```

long. Typically, for example, the fast track might present an idea and ask for a variation on the example shown; the normal track will first ask the student to repeat the example that was shown before attempting a variation. The third and slowest track, which is often three or four times the length of the fast track, is intended to be adequate for anyone. (The lessons of Figure 1 are from the third track.) The multiple tracks also mean that a student repeating a course is unlikely to hit the same series of lessons; this makes it profitable for a shaky user to back up and try again, and many students have done so.

The tracks are not completely distinct, however. Depending on the number of correct answers the student has given for the last few lessons, the program may switch tracks. The driver is actually capable of following an arbitrary directed graph of lesson sequences, as discussed in section 5. Some more structured arrangement, however, is used in all current scripts to aid the script writer in organizing the material into lessons. It is sufficiently difficult to write lessons that the three-track theory is not followed very closely except in the files and editor scripts. Accordingly, in some cases, the fast track is produced merely by skipping lessons from the slower track. In others, there is essentially only one track.

The main reason for using the *learn* program rather than simply writing the same material as a workbook is not the selection of tracks, but actual hands-on experience. Learning by doing is much more effective than pencil and paper exercises.

Learn also provides a mechanical check on performance. The first version in fact would not let the student proceed unless it received correct answers to the questions it set and it would not tell a student the right answer. This somewhat Draconian approach has been moderated in version 2. Lessons are sometimes badly worded or even just plain wrong; in such cases, the student has no recourse. But if a student is simply unable to complete one lesson, that should not prevent access to the rest. Accordingly, the current version of *learn* allows the student to skip a lesson that he cannot pass; a "no" answer to the "Do you want to try again?" question in Figure 1 will pass to the next lesson. It is still true that *learn* will not tell the student the right answer.

Of course, there are valid objections to the assumptions above. In particular, some students may object to not understanding what they are doing; and the procedure of smashing everything into small pieces may provoke the retort "you can't cross a ditch in two jumps." Since writing CAI scripts is considerably more tedious than ordinary manuals, however, it is safe to assume that there will always be alternatives to the scripts as a way of learning. In fact, for a reference manual of 3 or 4 pages it would not be surprising to have a tutorial manual of 20 pages and a (multi-track) script of 100 pages. Thus the reference manual will exist long before the scripts.

3. Scripts.

As mentioned above, the present scripts try at most to follow a three-track theory. Thus little of the potential complexity of the possible directed graph is employed, since care must be taken in lesson construction to see that every necessary fact is presented in every possible path through the units. In addition, it is desirable that every unit have alternate successors to deal with student errors.

In most existing courses, the first few lessons are devoted to checking prerequisites. For example, before the student is allowed to proceed through the editor script the script verifies that the student understands files and is able to type. It is felt that the sooner lack of student preparation is detected, the easier it will be on the student. Anyone proceeding through the scripts should be getting mostly correct answers; otherwise, the system will be unsatisfactory both because the wrong habits are being learned and because the scripts make little effort to deal with wrong answers. Unprepared students should not be encouraged to continue with scripts.

There are some preliminary items which the student must know before any scripts can be tried. In particular, the student must know how to connect to a UNIX system, set the terminal properly, log in, and execute simple commands (e.g., *learn* itself). In addition, the character erase and line kill conventions (# and @) should be known. It is hard to see how this much could be taught by computer-aided instruction, since a student who does not know these basic skills will not be able to run the learning program. A brief description on paper is provided (see Appendix A), although assistance will be needed for the first few minutes. This assistance, however, need not be highly skilled.

The first script in the current set deals with files. It assumes the basic knowledge above and teaches the student about the *ls*, *cat*, *mv*, *rm*, *cp* and *diff* commands. It also deals with the abbreviation characters *, ?, and [ ] in file names. It does not cover pipes or I/O redirection, nor does it present the many options on the *ls* command.

This script contains 31 lessons in the fast track; two are intended as prerequisite checks, seven are review exercises. There are a total of 75 lessons in all three tracks, and the instructional passages typed at the student to begin each lesson total 4,476 words. The average lesson thus begins with a 60-word message. In general, the fast track lessons have somewhat longer introductions, and the slow tracks somewhat shorter ones. The longest message is 144 words and the shortest 14.

The second script trains students in the use of the context editor *ed*, a sophisticated editor using regular expressions for searching.[5] All editor features except encryption, mark names and ';' in addressing are covered. The fast track contains 2 prerequisite checks, 93 lessons, and a review lesson. It is supplemented by 146 additional lessons in other tracks.

A comparison of sizes may be of interest. The *ed* description in the reference manual is 2,572 words long. The *ed* tutorial[6] is 6,138 words long. The fast track through the *ed* script is 7,407 words of explanatory messages, and the total *ed* script, 242 lessons, has 15,615 words. The average *ed* lesson is thus also about 60 words; the largest is 171 words and the smallest 10. The original *ed* script represents about three man-weeks of effort.

The advanced file handling script deals with *ls* options, I/O diversion, pipes, and supporting programs like *pr*, *wc*, *tail*, *spell* and *grep*. (The basic file handling script is a prerequisite.) It is not as refined as the first two scripts; this is reflected at least partly in the fact that it provides much less of a full three-track sequence than they do. On the other hand, since it is perceived as "advanced," it is hoped that the student will have somewhat more sophistication and be better able to cope with it at a reasonably high level of performance.

A fourth script covers the *eqn* language for typing mathematics. This script must be run on a terminal capable of printing mathematics, for instance the DASI 300 and similar Diablo-based terminals, or the nearly extinct Model 37 teletype. Again, this script is relatively short of tracks: of 76 lessons, only 17 are in the second track and 2 in the third track. Most of these provide additional practice for students who are having trouble in the first track.

The *−ms* script for formatting macros is a short one-track only script. The macro package it describes is no longer the standard, so this script will undoubtedly be superseded in the future. Furthermore, the linear style of a single learn script is somewhat inappropriate for the macros, since the macro package is composed of many independent features, and few users need all of them. It would be better to have a selection of short lesson sequences dealing with the features independently.

The script on C is in a state of transition. It was originally designed to follow a tutorial on C, but that document has since become obsolete. The current script has been partially converted to follow the order of presentation in *The C Programming Language*,[7] but this job is not complete. The C script was never intended to teach C; rather it is supposed to be a series of exercises for which the computer provides checking and (upon success) a suggested solution.

This combination of scripts covers much of the material which any user will need to know to make effective use of the UNIX system. With enlargement of the advanced files course to include more on the command interpreter, there will be a relatively complete introduction to UNIX available via *learn*. Although we make no pretense that *learn* will replace other instructional materials, it should provide a useful supplement to existing tutorials and reference manuals.

4. Experience with Students.

Learn has been installed on many different UNIX systems. Most of the usage is on the first two scripts, so these are more thoroughly debugged and polished. As a (random) sample of user experience, the *learn* program has been used at Bell Labs at Indian Hill for 10,500 lessons in a four month period. About 3600 of these are in the files script, 4100 in the editor, and 1400 in advanced files. The passing rate is about 80%, that is, about 4 lessons are passed for every one failed. There have been 86 distinct users of the files script, and 58 of the editor. On our system at Murray Hill, there have been nearly 4000 lessons over four weeks that include Christmas and New Year. Users have ranged in age from six up.

It is difficult to characterize typical sessions with the scripts; many instances exist of someone doing one or two lessons and then logging out, as do instances of someone pausing in a script for twenty minutes or more. In the earlier version of *learn*, the average session in the files course took 32 minutes and covered 23 lessons. The distribution is quite broad and skewed, however; the longest session was 130 minutes and there were five sessions shorter than five minutes. The average lesson took about 80 seconds. These numbers are roughly typical for non-programmers; a UNIX expert can [illegible] the scripts at approximately 30 seconds per lesson, most of which is the system printing.

At present working through a section of the middle of the files script took about 1.4 seconds of processor time per lesson, and a system expert typing quickly took 15 seconds of real time per lesson. A novice would probably take at least a minute. Thus, as a rough approximation, a UNIX system could support ten students working simultaneously with some spare capacity.

5. The Script Interpreter.

The *learn* program itself merely interprets scripts. It provides facilities for the script writer to capture student responses and their effects, and simplifies the job of passing control to and recovering control from the student. This section describes the operation and usage of the driver program, and indicates what is required to produce a new script. Readers only interested in the existing scripts may skip this section.

The file structure used by *learn* is shown in Figure 2. There is one parent directory (named *lib*) containing the script data. Within this directory are subdirectories, one for each subject in which a course is available, one for logging (named *log*), and one in which user subdirectories are created (named *play*). The subject directory contains master copies of all lessons, plus any supporting material for that subject. In a given subdirectory, each lesson is a single text file. Lessons are usually named systematically; the file that contains lesson *n* is called *Ln*.

When *learn* is executed, it makes a private directory for the user to work in, within the *learn* portion of the file system. A fresh copy of all the files used in each lesson (mostly data for the student to operate upon) is made each time a student starts a lesson, so the script writer may assume that everything is reinitialized each time a lesson is entered. The student directory is deleted after each session; any permanent records must be kept elsewhere.

The script writer must provide certain basic items in each lesson:

(1) the text of the lesson;

(2) the set-up commands to be executed before the user gets control;

(3) the data, if any, which the user is supposed to edit, transform, or otherwise process;

(4) the evaluating commands to be executed after the user has finished the lesson, to decide whether the answer is right; and

(5) a list of possible successor lessons.

Learn tries to minimize the work of bookkeeping and installation, so that most of the effort involved in script production is in planning lessons, writing tutorial paragraphs, and coding tests of student performance.

Figure 2: Directory structure for *learn*

```
lib
        play
                          student1
                                          files for student1...
                          student2
                                          files for student2...
        files
                          L0.1a           lessons for files course
                          L0.1b
                          ...
        editor
                          ...
        (other courses)
        log
```

The basic sequence of events is as follows. First, *learn* creates the working directory. Then, for each lesson, *learn* reads the script for the lesson and processes it a line at a time. The lines in the script are: (1) commands to the script interpreter to print something, to create a files, to test something, etc.; (2) text to be printed or put in a file; (3) other lines, which are sent to the shell to be executed. One line in each lesson turns control over to the user; the user can run any UNIX commands. The user mode terminates when the user types *yes*, *no*, *ready*, or *answer*. At this point, the user's work is tested; if the lesson is passed, a new lesson is selected, and if not the old one is repeated.

Let us illustrate this with the script for the second lesson of Figure 1; this is shown in Figure 3.

Lines which begin with # are commands to the *learn* script interpreter. For example,

#print

causes printing of any text that follows, up to the next line that begins with a sharp.

#print file

prints the contents of *file*; it is the same as *cat file* but has less overhead. Both forms of *#print* have the added property that if a lesson is failed, the *#print* will not be executed the second time through; this avoids annoying the student by repeating the preamble to a lesson.

#create filename

creates a file of the specified name, and copies any subsequent text up to a # to the file. This is used for creating and initializing working files and reference data for the lessons.

#user

gives control to the student; each line he or she types is passed to the shell for execution. The *#user* mode is terminated when the student types one of *yes*, *no*, *ready* or *answer*. At that time, the driver resumes interpretation of the script.

#copyin
#uncopyin

Anything the student types between these commands is copied onto a file called *.copy*. This lets the script writer interrogate the student's responses upon regaining control.

Figure 3: Sample Lesson

```
#print
Of course, you can print any file with "cat".
In particular, it is common to first use
"ls" to find the name of a file and then "cat"
to print it. Note the difference between
"ls", which tells you the name of the files,
and "cat", which tells you the contents.
One file in the current directory is named for
a President. Print the file, then type "ready".
#create roosevelt
  this file is named roosevelt
  and contains three lines of
  text.
#copyout
#user
#uncopyout
tail −3 .ocopy >X1
#cmp X1 roosevelt
#log
#next
3.2b 2
```

#copyout
#uncopyout

Between these commands, any material typed at the student by any program is copied to the file *.ocopy.* This lets the script writer interrogate the effect of what the student typed, which true believers in the performance theory of learning usually prefer to the student's actual input.

#pipe
#unpipe

Normally the student input and the script commands are fed to the UNIX command interpreter (the "shell") one line at a time. This won't do if, for example, a sequence of editor commands is provided, since the input to the editor must be handed to the editor, not to the shell. Accordingly, the material between *#pipe* and *#unpipe* commands is fed continuously through a pipe so that such sequences work. If *copyout* is also desired the *copyout* brackets must include the *pipe* brackets.

There are several commands for setting status after the student has attempted the lesson.

#cmp file1 file2

is an in-line implementation of *cmp*, which compares two files for identity.

#match stuff

The last line of the student's input is compared to *stuff*, and the success or fail status is set according to it. Extraneous things like the word *answer* are stripped before the comparison is made. There may be several *#match* lines; this provides a convenient mechanism for handling multiple "right" answers. Any text up to a # on subsequent lines after a successful *#match* is printed; this is illustrated in Figure 4, another sample lesson.

#bad stuff

This is similar to *#match*, except that it corresponds to specific failure answers; this can be used to produce hints for particular wrong answers that have been anticipated by the script

Figure 4: Another Sample Lesson

```
#print
What command will move the current line
to the end of the file? Type
"answer COMMAND", where COMMAND is the command.
#copyin
#user
#uncopyin
#match m$
#match .m$
"m$" is easier.
#log
#next
63.1d 10
```

writer.

#succeed
#fail

print a message upon success or failure (as determined by some previous mechanism).

When the student types one of the "commands" *yes*, *no*, *ready*, or *answer*, the driver terminates the *#user* command, and evaluation of the student's work can begin. This can be done either by the built-in commands above, such as *#match* and *#cmp*, or by status returned by normal UNIX commands, typically *grep* and *test*. The last command should return status true (0) if the task was done successfully and false (non-zero) otherwise; this status return tells the driver whether or not the student has successfully passed the lesson.

Performance can be logged:

#log file

writes the date, lesson, user name and speed rating, and a success/failure indication on *file*. The command

#log

by itself writes the logging information in the logging directory within the *learn* hierarchy, and is the normal form.

#next

is followed by a few lines, each with a successor lesson name and an optional speed rating on it. A typical set might read

```
25.1a  10
25.2a  5
25.3a  2
```

indicating that unit 25.1a is a suitable follow-on lesson for students with a speed rating of 10 units, 25.2a for student with speed near 5, and 25.3a for speed near 2. Speed ratings are maintained for each session with a student; the rating is increased by one each time the student gets a lesson right and decreased by four each time the student gets a lesson wrong. Thus the driver tries to maintain a level such that the users get 80% right answers. The maximum rating is limited to 10 and the minimum to 0. The initial rating is zero unless the student specifies a different rating when starting a session.

If the student passes a lesson, a new lesson is selected and the process repeats. If the student fails, a false status is returned and the program reverts to the previous lesson and tries another alternative. If it can not find another alternative, it skips forward a lesson. The student can terminate a session at any time by typing *bye*, which causes a graceful exit from *learn*. Hanging up is the usual novice's way out.

The lessons may form an arbitrary directed graph, although the present program imposes a limitation on cycles in that it will not present a lesson twice in the same session. If the student is unable to answer one of the exercises correctly, the driver searches for a previous lesson with a set of alternatives as successors (following the *#next* line). From the previous lesson with alternatives one route was taken earlier; the program simply tries a different one.

It is perfectly possible to write sophisticated scripts that evaluate the student's speed of response, or try to estimate the elegance of the answer, or provide detailed analysis of wrong answers. Lesson writing is so tedious already, however, that most of these abilities are likely to go unused.

The driver program depends heavily on features of the UNIX system that are not available on many other operating systems. These include the ease of manipulating files and directories, file redirection, the ability to use the command interpreter as just another program (even in a pipeline), command status testing and branching, the ability to catch signals like interrupts, and of course the pipeline mechanism itself. Although some parts of *learn* might be transferable to other systems, some generality will probably be lost.

A bit of history: The first version of *learn* had fewer built-in commands in the driver program, and made more use of the facilities of the UNIX system itself. For example, file comparison was done by creating a *cmp* process, rather than comparing the two files within *learn*. Lessons were not stored as text files, but as archives. There was no concept of the in-line document; even *#print* had to be followed by a file name. Thus the initialization for each lesson was to extract the archive into the working directory (typically 4-8 files), then *#print* the lesson text.

The combination of such things made *learn* rather slow and demanding of system resources. The new version is about 4 or 5 times faster, because fewer files and processes are created. Furthermore, it appears even faster to the user because in a typical lesson, the printing of the message comes first, and file setup with *#create* can be overlapped with printing, so that when the program finishes printing, it is really ready for the user to type at it.

It is also a great advantage to the script maintainer that lessons are now just ordinary text files, rather than archives. They can be edited without any difficulty, and UNIX text manipulation tools can be applied to them. The result has been that there is much less resistance to going in and fixing substandard lessons.

6. Conclusions

The following observations can be made about secretaries, typists, and other non-programmers who have used *learn*:

(a) A novice must have assistance with the mechanics of communicating with the computer to get through to the first lesson or two; once the first few lessons are passed people can proceed on their own.

(b) The terminology used in the first few lessons is obscure to those inexperienced with computers. It would help if there were a low level reference card for UNIX to supplement the existing programmer oriented bulky manual and bulky reference card.

(c) The concept of "substitutable argument" is hard to grasp, and requires help.

(d) They enjoy the system for the most part. Motivation matters a great deal, however.

It takes an hour or two for a novice to get through the script on file handling. The total time for a reasonably intelligent and motivated novice to proceed from ignorance to a reasonable ability to create new files and manipulate old ones seems to be a few days, with perhaps half of each day spent on the machine.

The normal way of proceeding has been to have students in the same room with someone who knows the UNIX system and the scripts. Thus the student is not brought to a halt by difficult questions. The burden on the counselor, however, is much lower than that on a teacher of a course. Ideally, the students should be encouraged to proceed with instruction immediately prior to their actual use of the computer. They should exercise the scripts on the same computer and the same kind of terminal that they will later use for their real work, and their first few jobs for the computer should be relatively easy ones. Also, both training and initial work should take place on days when the hardware and software are working reliably. Rarely is all of this possible, but the closer one comes the better the result. For example, if it is known that the hardware is shaky one day, it is better to attempt to reschedule training for another one. Students are very frustrated by machine downtime; when nothing is happening, it takes some sophistication and experience to distinguish an infinite loop, a slow but functioning program, a program waiting for the user, and a broken machine.*

One disadvantage of training with *learn* is that students come to depend completely on the CAI system, and do not try to read manuals or use other learning aids. This is unfortunate, not only because of the increased demands for completeness and accuracy of the scripts, but because the scripts do not cover all of the UNIX system. New users should have manuals (appropriate for their level) and read them; the scripts ought to be altered to recommend suitable documents and urge students to read them.

There are several other difficulties which are clearly evident. From the student's viewpoint, the most serious is that lessons still crop up which simply can't be passed. Sometimes this is due to poor explanations, but just as often it is some error in the lesson itself — a botched setup, a missing file, an invalid test for correctness, or some system facility that doesn't work on the local system in the same way it did on the development system. It takes knowledge and a certain healthy arrogance on the part of the user to recognize that the fault is not his or hers, but the script writer's. Permitting the student to get on with the next lesson regardless does alleviate this somewhat, and the logging facilities make it easy to watch for lessons that no one can pass, but it is still a problem.

The biggest problem with the previous *learn* was speed (or lack thereof) — it was often excruciatingly slow and a significant drain on the system. The current version so far does not seem to have that difficulty, although some scripts, notably *eqn*, are intrinsically slow. *eqn*, for example, must do a lot of work even to print its introductions, let alone check the student responses, but delay is perceptible in all scripts from time to time.

Another potential problem is that it is possible to break *learn* inadvertently, by pushing interrupt at the wrong time, or by removing critical files, or any number of similar slips. The defenses against such problems have steadily been improved, to the point where most students should not notice difficulties. Of course, it will always be possible to break *learn* maliciously, but this is not likely to be a problem.

One area is more fundamental — some commands are sufficiently global in their effect that *learn* currently does not allow them to be executed at all. The most obvious is *cd*, which changes to another directory. The prospect of a student who is learning about directories inadvertently moving to some random directory and removing files has deterred us from even writing lessons on *cd*, but ultimately lessons on such topics probably should be added.

7. Acknowledgments

We are grateful to all those who have tried *learn*, for we have benefited greatly from their suggestions and criticisms. In particular, M. E. Bittrich, J. L. Blue, S. I. Feldman, P. A. Fox, and M. J. McAlpin have provided substantial feedback. Conversations with E. Z. Rothkopf also provided many of the ideas in the system. We are also indebted to Don Jackowski for serving

* We have even known an expert programmer to decide the computer was broken when he had simply left his terminal in local mode. Novices have great difficulties with such problems.

as a guinea pig for the second version, and to Tom Plum for his efforts to improve the C script.

References

1. D. L. Bitzer and D. Skaperdas, "The Economics of a Large Scale Computer Based Education System: Plato IV," pp. 17-29 in *Computer Assisted Instruction, Testing and Guidance*, ed. Wayne Holtzman, Harper and Row, New York (1970).
2. D. C. Gray, J. P. Hulskamp, J. H. Kumm, S. Lichtenstein, and N. E. Nimmervoll, "COALA - A Minicomputer CAI System," *IEEE Trans. Education* E-20(1), pp.73-77 (Feb. 1977).
3. P. Suppes, "On Using Computers to Individualize Instruction," pp. 11-24 in *The Computer in American Education*, ed. D. D. Bushnell and D. W. Allen, John Wiley, New York (1967).
4. B. F. Skinner, "Why We Need Teaching Machines," *Harv. Educ. Review* **31**, pp.377-398, Reprinted in *Educational Technology*, ed. J. P. DeCecco, Holt, Rinehart & Winston (New York, 1964). (1961).
5. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Bell Laboratories (1978). See section *ed* (I).
6. B. W. Kernighan, *A tutorial introduction to the UNIX text editor*, Bell Laboratories internal memorandum (1974).
7. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

An Introduction to the UNIX Shell

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The *shell* is a command programming language that provides an interface to the UNIX† operating system. Its features include control-flow primitives, parameter passing, variables and string substitution. Constructs such as *while, if then else, case* and *for* are available. Two-way communication is possible between the *shell* and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as shell input.

The *shell* can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through 'pipes' can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file, which allows command procedures to be stored for later use.

November 12, 1978

†UNIX is a Trademark of Bell Laboratories.

An Introduction to the UNIX Shell

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

1.0 Introduction

The shell is both a command language and a programming language that provides an interface to the UNIX operating system. This memorandum describes, with examples, the UNIX shell. The first section covers most of the everyday requirements of terminal users. Some familiarity with UNIX is an advantage when reading this section; see, for example, "UNIX for beginners".[1] Section 2 describes those features of the shell primarily intended for use within shell procedures. These include the control-flow primitives and string-valued variables provided by the shell. A knowledge of a programming language would be a help when reading this section. The last section describes the more advanced features of the shell. References of the form "see *pipe* (2)" are to a section of the UNIX manual.[2]

1.1 Simple commands

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example,

```
who
```

is a command that prints the names of users logged in. The command

```
ls −l
```

prints a list of files in the current directory. The argument *−l* tells *ls* to print status information, size and the creation date for each file.

1.2 Background commands

To execute a command the shell normally creates a new *process* and waits for it to finish. A command may be run without waiting for it to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing & is an operator that instructs the shell not to wait for the command to finish. To help keep track of such a process the shell reports its process number following its creation. A list of currently active processes may be obtained using the *ps* command.

1.3 Input output redirection

Most commands produce output on the standard output that is initially connected to the terminal. This output may be sent to a file by writing, for example,

```
ls −l >file
```

The notation *>file* is interpreted by the shell and is not passed as an argument to *ls*. If *file* does not exist then the shell creates it; otherwise the original contents of *file* are replaced with the output from *ls*. Output may be appended to a file using the notation

```
ls -l >>file
```

In this case *file* is also created if it does not already exist.

The standard input of a command may be taken from a file instead of the terminal by writing, for example,

```
wc <file
```

The command *wc* reads its standard input (in this case redirected from *file*) and prints the number of characters, words and lines found. If only the number of lines is required then

```
wc -l <file
```

could be used.

1.4 Pipelines and filters

The standard output of one command may be connected to the standard input of another by writing the 'pipe' operator, indicated by |, as in,

```
ls -l | wc
```

Two commands connected in this way constitute a *pipeline* and the overall effect is the same as

```
ls -l >file; wc <file
```

except that no *file* is used. Instead the two processes are connected by a pipe (see *pipe* (2)) and are run in parallel. Pipes are unidirectional and synchronization is achieved by halting *wc* when there is nothing to read and halting *ls* when the pipe is full.

A *filter* is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, *grep*, selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines, if any, of the output from *ls* that contain the string *old*. Another useful filter is *sort*. For example,

```
who | sort
```

will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

```
ls | grep old | wc -l
```

prints the number of file names in the current directory containing the string *old*.

1.5 File name generation

Many commands accept arguments which are file names. For example,

```
ls -l main.c
```

prints information relating to the file *main.c*.

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates, as arguments to *ls*, all file names in the current directory that end in *.c*. The character * is a pattern that will match any string including the null string. In general *patterns* are specified as follows.

*	Matches any string of characters including the null string.
?	Matches any single character.
[. . .]	Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
[a–z]*
```

matches all names in the current directory beginning with one of the letters *a* through *z*.

```
/usr/fred/test/?
```

matches all names in the directory /usr/fred/test that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all *core* files in sub-directories of **/usr/fred**. (*echo* is a standard UNIX command that prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all sub-directories of **/usr/fred**.

There is one exception to the general rules given for patterns. The character '.' at the start of a file name must be explicitly matched.

```
echo *
```

will therefore echo all file names in the current directory not beginning with '.'.

```
echo .*
```

will echo all those file names that begin with '.'. This avoids inadvertent matching of the names '.' and '..' which mean 'the current directory' and 'the parent directory' respectively. (Notice that *ls* suppresses information for the files '.' and '..'.)

1.6 Quoting

Characters that have a special meaning to the shell, such as < > * ? | &, are called metacharacters. A complete list of metacharacters is given in appendix B. Any character preceded by a \ is *quoted* and loses its special meaning, if any. The \ is elided so that

```
echo \?
```

will echo a single ?, and

```
echo \\
```

will echo a single \. To allow long strings to be continued over more than one line the sequence **newline** is ignored.

\ is convenient for quoting single characters. When more than one character needs quoting the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

```
echo xx'****'xx
```

will echo

```
xx****xx
```

The quoted string may not contain a single quote but may contain newlines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use.

A third quoting mechanism using double quotes is also available that prevents interpretation of some but not all metacharacters. Discussion of the details is deferred to section 3.4.

1.7 Prompting

When the shell is used from a terminal it will issue a prompt before reading a command. By default this prompt is '$ '. It may be changed by saying, for example,

```
PS1=yesdear
```

that sets the prompt to be the string *yesdear*. If a newline is typed and further input is needed then the shell will issue the prompt '> '. Sometimes this can be caused by mistyping a quote mark. If it is unexpected then an interrupt (DEL) will return the shell to read another command. This prompt may be changed by saying, for example,

```
PS2=more
```

1.8 The shell and login

Following *login* (1) the shell is called to read and execute commands typed at the terminal. If the user's login directory contains the file **.profile** then it is assumed to contain commands and is read by the shell before reading any commands from the terminal.

1.9 Summary

- **ls**
  Print the names of files in the current directory.
- **ls >file**
  Put the output from *ls* into *file*.
- **ls | wc −l**
  Print the number of files in the current directory.
- **ls | grep old**
  Print those file names containing the string *old*.
- **ls | grep old | wc −l**
  Print the number of files whose name contains the string *old*.
- **cc pgm.c &**
  Run *cc* in the background.

2.0 Shell procedures

The shell may be used to read and execute commands contained in a file. For example,

```
sh file [ args ... ]
```

calls the shell to read commands from *file*. Such a file is called a *command procedure* or *shell procedure*. Arguments may be supplied with the call and are referred to in *file* using the positional parameters $1, $2, ... For example, if the file *wg* contains

```
who | grep $1
```

then

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

UNIX files have three independent attributes, *read*, *write* and *execute*. The UNIX command *chmod* (1) may be used to make a file executable. For example,

```
chmod +x wg
```

will ensure that the file *wg* has execute status. Following this, the command

```
wg fred
```

is equivalent to

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably. In either case a new process is created to run the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as $# . The name of the file being executed is available as $0 .

A special shell parameter $* is used to substitute for all positional parameters except $0 . A typical use of this is to provide some default arguments, as in,

```
nroff -T450 -ms $*
```

which simply prepends some arguments to those already given.

2.1 Control flow - for

A frequent use of shell procedures is to loop through the arguments ($1, $2, ...) executing commands once for each argument. An example of such a procedure is *tel* that searches the file /usr/lib/telnos that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *tel* is

```
for i
do grep $i /usr/lib/telnos; done
```

The command

```
tel fred
```

prints those lines in /usr/lib/telnos that contain the string *fred*.

```
        tel fred bert
```

prints those lines containing *fred* followed by those for *bert*.

The **for** loop notation is recognized by the shell and has the general form

> **for** *name* **in** *w1 w2* ...
> **do** *command-list*
> **done**

A *command-list* is a sequence of one or more simple commands separated or terminated by a newline or semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a newline or semicolon. *name* is a shell variable that is set to the words *w1 w2* ... in turn each time the *command-list* following **do** is executed. If **in** *w1 w2* ... is omitted then the loop is executed once for each positional parameter; that is, **in** *$** is assumed.

Another example of the use of the **for** loop is the *create* command whose text is

```
        for i do >$i; done
```

The command

```
        create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. The notation >*file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or newline) is required before **done**.

2.2 Control flow - case

A multiple way branch is provided for by the **case** notation. For example,

```
        case $# in
             1) cat >>$1 ;;
             2) cat >>$2 <$1 ;;
             *) echo 'usage: append [ from ] to' ;;
        esac
```

is an *append* command. When called with one argument as

```
        append file
```

$# is the string *1* and the standard input is copied onto the end of *file* using the *cat* command.

```
        append file1 file2
```

appends the contents of *file1* onto *file2*. If the number of arguments supplied to *append* is other than 1 or 2 then a message is printed indicating proper usage.

The general form of the **case** command is

> **case** *word* **in**
>   *pattern*) *command-list* ;;
>   ...
> **esac**

The shell attempts to match *word* with each *pattern*, in the order in which the patterns appear. If a match is found the associated *command-list* is executed and execution of the case is complete. Since * is the pattern that matches any string it can be used for the default case.

A word of caution: no check is made to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed. In the example below the commands following the second * will never be executed.

```
case $# in
     *) . . . ;;
     *) . . . ;;
esac
```

Another example of the use of the case construction is to distinguish between different forms of an argument. The following example is a fragment of a *cc* command.

```
for i
do case $i in
   −[ocs])      . . . ;;
   −*) echo 'unknown flag $i' ;;
   *.c) /lib/c0 $i . . . ;;
   *)    echo 'unexpected argument $i' ;;
   esac
done
```

To allow the same commands to be associated with more than one pattern the case command provides for alternative patterns separated by a | . For example,

```
case $i in
     −x|−y)    . . .
esac
```

is equivalent to

```
case $i in
     −[xy])    . . .
esac
```

The usual quoting conventions apply so that

```
case $i in
     \?)       . . .
```

will match the character ? .

2.3 Here documents

The shell procedure *tel* in section 2.1 uses the file /usr/lib/telnos to supply the data for *grep*. An alternative is to include this data within the shell procedure as a *here* document, as in,

```
for i
do grep $i <<!
   . . .
   fred mh0123
   bert mh0789
   . . .
!
done
```

In this example the shell takes the lines between <<! and ! as the standard input for *grep*. The string ! is arbitrary, the document being terminated by a line that consists of the string following << .

Parameters are substituted in the document before it is made available to *grep* as illustrated by the following procedure called *edg*.

```
ed $3 <<%
g/$1/s//$2/g
w
%
```

The call

```
edg string1 string2 file
```

is then equivalent to the command

```
ed file <<%
g/string1/s//string2/g
w
%
```

and changes all occurrences of *string1* in *file* to *string2*. Substitution can be prevented using \ to quote the special character **$** as in

```
ed $3 <<+
1,\$s/$1/$2/g
w
+
```

(This version of *edg* is equivalent to the first except that *ed* will print a **?** if there are no occurrences of the string **$1**.) Substitution within a *here* document may be prevented entirely by quoting the terminating string, for example,

```
grep $i <<\#
...
#
```

The document is presented without modification to *grep*. If parameter substitution is not required in a *here* document this latter form is more efficient.

2.4 Shell variables

The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits and underscores. Variables may be given values by writing, for example,

```
user=fred box=m000 acct=mh0000
```

which assigns values to the variables **user**, **box** and **acct**. A variable may be set to the null string by saying, for example,

```
null=
```

The value of a variable is substituted by preceding its name with **$**; for example,

```
echo $user
```

will echo *fred*.

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv pgm $b
```

will move the file *pgm* from the current directory to the directory **/usr/fred/bin**. A more general notation is available for parameter (or variable) substitution, as in,

```
echo ${user}
```

which is equivalent to

```
        echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
        tmp=/tmp/ps
        ps a >${tmp}a
```

will direct the output of *ps* to the file /tmp/psa, whereas,

```
        ps a >$tmpa
```

would cause the value of the variable **tmpa** to be substituted.

Except for **$?** the following are set initially by the shell. **$?** is set after executing each command.

$? The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non-zero exit status is returned. Testing the value of return codes is dealt with later under **if** and **while** commands.

$# The number of positional parameters (in decimal). Used, for example, in the *append* command to check the number of parameters.

$$ The process number of this shell (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,

```
        ps a >/tmp/ps$$
        ...
        rm /tmp/ps$$
```

$! The process number of the last process run in the background (in decimal).

$– The current shell flags, such as −x and −v.

Some variables have a special meaning to the shell and should be avoided for general use.

$MAIL When used interactively the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at the shell prints the message *you have mail* before prompting for the next command. This variable is typically set in the file **.profile**, in the user's login directory. For example,

```
        MAIL=/usr/mail/fred
```

$HOME The default argument for the *cd* command. The current directory is used to resolve file name references that do not begin with a /, and is changed using the *cd* command. For example,

```
        cd /usr/fred/bin
```

makes the current directory **/usr/fred/bin**.

```
        cat wn
```

will print on the terminal the file *wn* in this directory. The command *cd* with no argument is equivalent to

```
        cd $HOME
```

This variable is also typically set in the the user's login profile.

$PATH A list of directories that contain commands (the *search path*). Each time a command is executed by the shell a list of directories is searched for an executable file. If **$PATH** is not set then the current directory, **/bin**, and **/usr/bin** are searched by default. Otherwise **$PATH** consists of directory names separated by **:**. For example,

```
PATH=:/usr/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first **:**), **/usr/fred/bin**, **/bin** and **/usr/bin** are to be searched in that order. In this way individual users can have their own 'private' commands that are accessible independently of the current directory. If the command name contains a / then this directory search is not used; a single attempt is made to execute the command.

$PS1	The primary shell prompt string, by default, '$ '.
$PS2	The shell prompt when further input is needed, by default, '> '.
$IFS	The set of characters used by *blank interpretation* (see section 3.4).

2.5 The test command

The *test* command, although not part of the shell, is intended for use by shell programs. For example,

```
test −f file
```

returns zero exit status if *file* exists and non-zero exit status otherwise. In general *test* evaluates a predicate and returns the result as its exit status. Some of the more frequently used *test* arguments are given here, see *test* (1) for a complete specification.

test s	true if the argument *s* is not the null string
test −f file	true if *file* exists
test −r file	true if *file* is readable
test −w file	true if *file* is writable
test −d file	true if *file* is a directory

2.6 Control flow - while

The actions of the **for** loop and the **case** branch are determined by data available to the shell. A **while** or **until** loop and an **if then else** branch are also provided whose actions are determined by the exit status returned by commands. A **while** loop has the general form

while *command-list*$_1$
do *command-list*$_2$
done

The value tested by the **while** command is the exit status of the last simple command following **while**. Each time round the loop *command-list*$_1$ is executed; if a zero exit status is returned then *command-list*$_2$ is executed; otherwise, the loop terminates. For example,

```
while test $1
do ...
   shift
done
```

is equivalent to

```
for i
do ...
done
```

shift is a shell command that renames the positional parameters **$2, $3, ...** as **$1, $2, ...** and loses **$1**.

Another kind of use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop the termination condition is reversed. For example,

```
until test -f file
do sleep 300; done
commands
```

will loop until *file* exists. Each time round the loop it waits for 5 minutes before trying again. (Presumably another process will eventually create the file.)

2.7 Control flow - if

Also available is a general conditional branch of the form,

```
if command-list
then    command-list
else    command-list
fi
```

that tests the value returned by the last simple command following **if**.

The **if** command may be used in conjunction with the *test* command to test for the existence of a file as in

```
if test -f file
then    process file
else    do something else
fi
```

An example of the use of **if**, **case** and **for** constructions is given in section 2.10.

A multiple test **if** command of the form

```
if ...
then    ...
else    if ...
        then    ...
        else    if ...
                ...
                fi
        fi
fi
```

may be written using an extension of the **if** notation as,

```
if ...
then    ...
elif    ...
then    ...
elif    ...
...
fi
```

The following example is the *touch* command which changes the 'last modified' time for a list of files. The command may be used in conjunction with *make* (1) to force recompilation of a list of files.

```
flag=
for i
do case $i in
   -c) flag=N ;;
   *)  if test -f $i
       then    ln $i junk$$; rm junk$$
       elif test $flag
       then    echo file \'$i\' does not exist
       else    >$i
       fi
   esac
done
```

The −c flag is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable *flag* is set to some non-null string if the −c argument is encountered. The commands

```
ln ...; rm ...
```

make a link to the file and then remove it thus causing the last modified date to be updated.

The sequence

```
if command1
then    command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes *command2* only if *command1* fails. In each case the value returned is that of the last simple command executed.

2.8 Command grouping

Commands may be grouped in two ways,

{ *command-list* ; }

and

(*command-list*)

In the first *command-list* is simply executed. The second form executes *command-list* as a separate process. For example,

```
(cd x; rm junk )
```

executes *rm junk* in the directory x without changing the current directory of the invoking shell.

The commands

```
cd x; rm junk
```

have the same effect but leave the invoking shell in the directory x.

2.9 Debugging shell procedures

The shell provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

```
set −v
```

(v for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by saying

```
sh −v proc ...
```

where *proc* is the name of the shell procedure. This flag may be used in conjunction with the **−n** flag which prevents execution of subsequent commands. (Note that saying *set −n* at a terminal will render the terminal useless until an end-of-file is typed.)

The command

```
set −x
```

will produce an execution trace. Following parameter substitution each command is printed as it is executed. (Try these at the terminal to see what effect they have.) Both flags may be turned off by saying

```
set −
```

and the current setting of the shell flags is available as **$−** .

2.10 The man command

The following is the *man* command which is used to print sections of the UNIX manual. It is called, for example, as

```
man sh
man −t ed
man 2 fork
```

In the first the manual section for *sh* is printed. Since no section is specified, section 1 is used. The second example will typeset (−t option) the manual section for *ed*. The last prints the *fork* manual page from section 2.

```
cd /usr/man

: 'colon is the comment command'
: 'default is nroff ($N), section 1 ($s)'
N=n s=1

for i
do case $i in

    [1-9]*)     s=$i ;;

    -t)  N=t ;;

    -n)  N=n ;;

    -*)  echo unknown flag \'$i\' ;;

    *)   if test -f man$s/$i.$s
         then    ${N}roff man0/${N}aa man$s/$i.$s
         else    : 'look through all manual sections'
                 found=no
                 for j in 1 2 3 4 5 6 7 8 9
                 do if test -f man$j/$i.$j
                    then man $j $i
                         found=yes
                    fi
                 done
                 case $found in
                      no) echo '$i: manual page not found'
                 esac
         fi
    esac
done
```

Figure 1. A version of the man command

3.0 Keyword parameters

Shell variables may be given values by assignment or when a shell procedure is invoked. An argument to a shell procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking shell is not affected. For example,

```
user=fred command
```

will execute *command* with user set to *fred.* The −k flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain they are available as positional parameters $1, $2, ...

The *set* command may also be used to set positional parameters from within a procedure. For example,

```
set − *
```

will set $1 to the first file name in the current directory, $2 to the next, and so on. Note that the first argument, −, ensures correct treatment when the first file name begins with a −.

3.1 Parameter transmission

When a shell procedure is invoked both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a shell procedure by specifying in advance that such parameters are to be exported. For example,

```
export user box
```

marks the variables user and box for export. When a shell procedure is invoked copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking shell. It is generally true of a shell procedure that it may not modify the state of its caller without explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared *readonly*. The form of this command is the same as that of the *export* command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

3.2 Parameter substitution

If a shell parameter is not set then the null string is substituted for it. For example, if the variable d is not set

```
echo $d
```

or

```
echo ${d}
```

will echo nothing. A default string may be given as in

```
echo ${d−.}
```

which will echo the value of the variable d if it is set and '.' otherwise. The default string is evaluated using the usual quoting conventions so that

```
echo ${d−'*'}
```

will echo * if the variable d is not set. Similarly

```
echo ${d−$1}
```

will echo the value of **d** if it is set and the value (if any) of **$1** otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.}
```

which substitutes the same string as

```
echo ${d−.}
```

and if **d** were not previously set then it will be set to the string '.'. (The notation ${...=...} is not available for positional parameters.)

If there is no sensible default then the notation

```
echo ${d?message}
```

will echo the value of the variable **d** if it has one, otherwise *message* is printed by the shell and execution of the shell procedure is abandoned. If *message* is absent then a standard message is printed. A shell procedure that requires some parameters to be set might start as follows.

```
: ${user?} ${acct?} ${bin?}
...
```

Colon (:) is a command that is built in to the shell and does nothing once its arguments have been evaluated. If any of the variables **user, acct** or **bin** are not set then the shell will abandon execution of the procedure.

3.3 Command substitution

The standard output from a command can be substituted in a similar way to parameters. The command *pwd* prints on its standard output the name of the current directory. For example, if the current directory is **/usr/fred/bin** then the command

```
d=`pwd`
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between grave accents (`...`) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ` must be escaped using a \. For example,

```
ls `echo "$1"`
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including *here* documents) and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within shell procedures. An example of such a command is *basename* which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string *main*. Its use is illustrated by the following fragment from a *cc* command.

```
case $A in
    ...
    *.c)        B=`basename $A .c`
    ...
esac
```

that sets **B** to the part of **$A** with the suffix **.c** stripped.

Here are some composite examples.

- **for i in `ls −t`; do ...**
  The variable **i** is set to the names of files in time order, most recent first.
- **set `date`; echo $6 $2 $3, $4**
  will print, e.g., *1977 Nov 1, 23:59:59*

3.4 Evaluation and quoting

The shell is a macro processor that provides parameter substitution, command substitution and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in appendix A. Before a command is executed the following substitutions occur.

- parameter substitution, e.g. **$user**
- command substitution, e.g. **`pwd`**

  Only one evaluation occurs so that if, for example, the value of the variable **X** is the string *$y* then

  ```
  echo $X
  ```

  will echo *$y*.
- blank interpretation

  Following the above substitutions the resulting characters are broken into non-blank words (*blank interpretation*). For this purpose 'blanks' are the characters of the string **$IFS**. By default, this string consists of blank, tab and newline. The null string is not regarded as a word unless it is quoted. For example,

  ```
  echo ''
  ```

  will pass on the null string as the first argument to *echo*, whereas

  ```
  echo $null
  ```

  will call *echo* with no arguments if the variable **null** is not set or set to the null string.
- file name generation

  Each word is then scanned for the file pattern characters *, ? and [...] and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a **for** loop. Only substitution occurs in the *word* used for a case branch.

As well as the quoting mechanisms described earlier using \ and '...' a third quoting mechanism is provided using double quotes. Within double quotes parameter and command substitution occurs but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using \.

$	parameter substitution
`	command substitution
"	ends the quoted string
\	quotes the special characters $ ` " \

For example,

```
echo "$x"
```

will pass the value of the variable x as a single argument to *echo*. Similarly,

```
echo "$*"
```

will pass the positional parameters as a single argument and is equivalent to

```
echo "$1 $2 ..."
```

The notation $@ is the same as $* except when it is quoted.

```
echo "$@"
```

will pass the positional parameters, unevaluated, to *echo* and is equivalent to

```
echo "$1" "$2" ...
```

The following table gives, for each quoting mechanism, the shell metacharacters that are evaluated.

	metacharacter					
	\	$	*	`	"	'
'	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

t	terminator
y	interpreted
n	not interpreted

Figure 2. Quoting mechanisms

In cases where more than one evaluation of a string is required the built-in command *eval* may be used. For example, if the variable X has the value *$y*, and if y has the value *pqr* then

```
eval echo $X
```

will echo the string *pqr*.

In general the *eval* command evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed. For example,

```
wg='eval who|grep'
$wg fred
```

is equivalent to

```
who|grep fred
```

In this example, *eval* is required since there is no interpretation of metacharacters, such as | , following substitution.

3.5 Error handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a terminal (as determined by *gtty* (2)). A shell invoked with the **−i** flag is also interactive.

Execution of a command (see also 3.7) may fail for any of the following reasons.

- Input output redirection may fail. For example, if a file does not exist or cannot be created.

- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a "bus error" or "memory fault". See Figure 2 below for a complete list of UNIX signals.
- The command terminates normally but returns a non-zero exit status.

In all of these cases the shell will go on to execute the next command. Except for the last case an error message will be printed by the shell. All remaining errors cause the shell to exit from a command procedure. An interactive shell will return to read another command from the terminal. Such errors include the following.

- Syntax errors. e.g., if ... then ... done
- A signal such as interrupt. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as *cd.*

The shell flag −e causes the shell to terminate if any error is detected.

1	hangup
2	interrupt
3*	quit
4*	illegal instruction
5*	trace trap
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination (from *kill* (1))

Figure 3. UNIX signals

Those signals marked with an asterisk produce a core dump if not caught. However, the shell itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to shell programs are 1, 2, 3, 14 and 15.

3.6 Fault handling

Shell procedures normally terminate when an interrupt is received from the terminal. The *trap* command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt), and if this signal is received will execute the commands

```
rm /tmp/ps$$; exit
```

exit is another built-in command that terminates execution of a shell procedure. The *exit* is required; otherwise, after the trap has been taken, the shell will resume executing the procedure at the place where it was interrupted.

UNIX signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. Lastly, they can be left to cause termination of the process without it having to take any further action. If a signal is being ignored on entry to the shell procedure, for example, by invoking it in the background (see 3.7) then *trap* commands (and the signal) are ignored.

The use of *trap* is illustrated by this modified version of the *touch* command (Figure 4). The cleanup action is to remove the file **junk$$**.

```
flag=
trap 'rm −f junk$$; exit' 1 2 3 15
for i
do case $i in
    −c) flag=N ;;
    *)  if test −f $i
        then    ln $i junk$$; rm junk$$
        elif test $flag
        then    echo file \'$i\' does not exist
        else    >$i
        fi
    esac
done
```

Figure 4. The touch command

The *trap* command appears before the creation of the temporary file; otherwise it would be possible for the process to die without removing the file.

Since there is no signal 0 in UNIX it is used by the shell to indicate the commands to be executed on exit from the shell procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following fragment is taken from the *nohup* command.

```
trap '' 1 2 3 15
```

which causes *hangup, interrupt, quit* and *kill* to be ignored both by the procedure and by invoked commands.

Traps may be reset by saying

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```

The procedure *scan* (Figure 5) is an example of the use of *trap* where there is no exit in the trap command. *scan* takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when *scan* is waiting for input.

```
d=`pwd`
for i in *
do if test -d $d/$i
   then cd $d/$i
        while echo "$i:"
              trap exit 2
              read x
        do trap : 2; eval $x; done
   fi
done
```

Figure 5. The scan command

read x is a built-in command that reads one line from the standard input and places the result in the variable **x**. It returns a non-zero exit status if either an end-of-file is read or an interrupt is received.

3.7 Command execution

To run a command (other than a built-in) the shell first creates a new process using the system call *fork*. The execution environment for the command includes input, output and the states of signals, and is established in the child process before the command is executed. The built-in command *exec* is used in the rare cases when no fork is required and simply replaces the shell with a new command. For example, a simple version of the *nohup* command looks like

```
trap '' 1 2 3 15
exec $*
```

The *trap* turns off the signals specified so that they are ignored by subsequently created commands and *exec* replaces the shell by the command specified.

Most forms of input output redirection have already been described. In the following *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ... >*.c
```

will write its output into a file whose name is ***.c**. Input output specifications are evaluated left to right as they appear in the command.

> *word*	The standard output (file descriptor 1) is sent to the file *word* which is created if it does not already exist.
>> *word*	The standard output is sent to file *word*. If the file exists then output is appended (by seeking to the end); otherwise the file is created.
< *word*	The standard input (file descriptor 0) is taken from the file *word*.
<< *word*	The standard input is taken from the lines of shell input that follow up to but not including a line consisting only of *word*. If *word* is quoted then no interpretation of the document occurs. If *word* is not quoted then parameter and command substitution occur and \ is used to quote the characters \ $ ` and the first character of *word*. In the latter case **\newline** is ignored (c.f. quoted strings).
>& *digit*	The file descriptor *digit* is duplicated using the system call *dup* (2) and the result is used as the standard output.
<& *digit*	The standard input is duplicated from file descriptor *digit*.

<&− The standard input is closed.

>&− The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to *file*.

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking file descriptor 2 is created by duplicating file descriptor 1 but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. Firstly, the default standard input for such a command is the empty file **/dev/null**. This prevents two processes (the shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
ed file &
```

would allow both the editor and the shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason the UNIX convention for a signal is that if it is set to 1 (ignored) then it is never changed even for a short time. Note that the shell command *trap* has no effect for an ignored signal.

3.8 Invoking the shell

The following flags are interpreted by the shell when it is invoked. If the first character of argument zero is a minus, then commands are read from the file **.profile**.

−c *string*
 If the **−c** flag is present then commands are read from *string*.

−s If the **−s** flag is present or if no arguments remain then commands are read from the standard input. Shell output is written to file descriptor 2.

−i If the **−i** flag is present or if the shell input and output are attached to a terminal (as told by *gtty*) then this shell is *interactive*. In this case TERMINATE is ignored (so that **kill 0** does not kill an interactive shell) and INTERRUPT is caught and ignored (so that **wait** is interruptable). In all cases QUIT is ignored by the shell.

Acknowledgements

The design of the shell is based in part on the original UNIX shell[3] and the PWB/UNIX shell,[4] some features having been taken from both. Similarities also exist with the command interpreters of the Cambridge Multiple Access System[5] and of CTSS.[6]

I would like to thank Dennis Ritchie and John Mashey for many discussions during the design of the shell. I am also grateful to the members of the Computing Science Research Center and to Joe Maranzano for their comments on drafts of this document.

References

1. B. W. Kernighan, *UNIX for Beginners*, Bell Laboratories internal memorandum (1978).
2. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Bell Laboratories (1978). Seventh Edition.
3. K. Thompson, "The UNIX Command Language," pp. 375-384 in *Structured Programming—Infotech State of the Art Report*, Infotech International Ltd., Nicholson House, Maidenhead, Berkshire, England (March 1975).
4. J. R. Mashey, *PWB/UNIX Shell Tutorial*, Bell Laboratories internal memorandum (September 30, 1977).
5. D. F. Hartley (Ed.), *The Cambridge Multiple Access System — Users Reference Manual*, University Mathematical Laboratory, Cambridge, England (1968).
6. P. A. Crisman (Ed.), *The Compatible Time-Sharing System*, M.I.T. Press, Cambridge, Mass. (1965).

Appendix A - Grammar

item:	*word*
	input-output
	name = *value*
simple-command:	*item*
	simple-command item
command:	*simple-command*
	(*command-list*)
	{ *command-list* }
	for *name* **do** *command-list* **done**
	for *name* **in** *word* **...** **do** *command-list* **done**
	while *command-list* **do** *command-list* **done**
	until *command-list* **do** *command-list* **done**
	case *word* **in** *case-part* **...** **esac**
	if *command-list* **then** *command-list else-part* **fi**
pipeline:	*command*
	pipeline | *command*
andor:	*pipeline*
	andor && *pipeline*
	andor || *pipeline*
command-list:	*andor*
	command-list ;
	command-list &
	command-list ; *andor*
	command-list & *andor*
input-output:	> *file*
	< *file*
	>> *word*
	<< *word*
file:	*word*
	& *digit*
	& −
case-part:	*pattern*) *command-list* ;;
pattern:	*word*
	pattern | *word*
else-part:	**elif** *command-list* **then** *command-list else-part*
	else *command-list*
	empty
empty:	
word:	a sequence of non-blank characters
name:	a sequence of letters, digits or underscores starting with a letter
digit:	**0 1 2 3 4 5 6 7 8 9**

Appendix B - Meta-characters and Reserved Words

a) syntactic

|	pipe symbol
&&	'andf' symbol
||	'orf' symbol
;	command separator
;;	case delimiter
&	background commands
( )	command grouping
<	input redirection
<<	input from a here document
>	output creation
>>	output append

b) patterns

*	match any character(s) including none
?	match any single character
[...]	match any of the enclosed characters

c) substitution

${...}	substitute shell variable
\`...\`	substitute command output

d) quoting

\	quote the next character
'...'	quote the enclosed characters except for '
"..."	quote the enclosed characters except for $ \` \ "

e) reserved words

if then else elif fi
case in esac
for while until do done
{ }

An introduction to the C shell

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

Csh is a new command language interpreter for UNIX† systems. It incorporates good features of other shells and a *history* mechanism similar to the *redo* of INTERLISP. While incorporating many features of other shells which make writing shell programs (shell scripts) easier, most of the features unique to *csh* are designed more for the interactive UNIX user.

UNIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with *csh* is possible after reading just the first section of this document. The second section describes the shells capabilities which you can explore after you have begun to become acquainted with the shell. Later sections introduce features which are useful, but not necessary for all users of the shell.

Back matter includes an appendix listing special characters of the shell and a glossary of terms and commands introduced in this manual.

November 8, 1980

†UNIX is a Trademark of Bell Laboratories.

An introduction to the C shell

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Introduction

A *shell* is a command language interpreter. *Csh* is the name of one particular command interpreter on UNIX. The primary purpose of *csh* is to translate command lines typed at a terminal into system actions, such as invocation of other programs. *Csh* is a user program just like any you might write. Hopefully, *csh* will be a very useful program for you in interacting with the UNIX system.

In addition to this document, you will want to refer to a copy of the UNIX programmer's manual. The *csh* documentation in the manual provides a full description of all features of the shell and is a final reference for questions about the shell.

Many words in this document are shown in *italics*. These are important words; names of commands, and words which have special meaning in discussing the shell and UNIX. Many of the words are defined in a glossary at the end of this document. If you don't know what is meant by a word, you should look for it in the glossary.

Acknowledgements

Numerous people have provided good input about previous versions of *csh* and aided in its debugging and in the debugging of its documentation. I would especially like to thank Michael Ubell who made the crucial observation that history commands could be done well over the word structure of input text, and implemented a prototype history mechanism in an older version of the shell. Eric Allman has also provided a large number of useful comments on the shell, helping to unify those concepts which are present and to identify and eliminate useless and marginally useful features. Mike O'Brien suggested the pathname hashing mechanism which speeds command execution. Jim Kulp added the job control and directory stack primitives and added their documentation to this introduction.

1. Terminal usage of the shell

1.1. The basic notion of commands

A *shell* in UNIX acts mostly as a medium through which other *programs* are invoked. While it has a set of *builtin* functions which it performs directly, most commands cause execution of programs that are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

Commands in the UNIX system consist of a list of strings or *words* interpreted as a *command name* followed by *arguments.* Thus the command

```
mail bill
```

consists of two words. The first word *mail* names the command to be executed, in this case the mail program which sends messages to other users. The shell uses the name of the command in attempting to execute it for you. It will look in a number of *directories* for a file with the name *mail* which is expected to contain the mail program.

The rest of the words of the command are given as *arguments* to the command itself when it is executed. In this case we specified also the argument *bill* which is interpreted by the *mail* program to be the name of a user to whom mail is to be sent. In normal terminal usage we might use the *mail* command as follows.

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page 5.
Does a page five exist?
        Bill
EOT
%
```

Here we typed a message to send to *bill* and ended this message with a ↑D which sent an end-of-file to the mail program. (Here and throughout this document, the notation "↑*x*" is to be read "control-*x*" and represents the striking of the *x* key while the control key is held down.) The mail program then echoed the characters 'EOT' and transmitted our message. The characters '% ' were printed before and after the mail command by the shell to indicate that input was needed.

After typing the '% ' prompt the shell was reading command input from our terminal. We typed a complete command 'mail bill'. The shell then executed the *mail* program with argument *bill* and went dormant waiting for it to complete. The mail program then read input from our terminal until we signalled an end-of-file via typing a ↑D after which the shell noticed that mail had completed and signaled us that it was ready to read from the terminal again by printing another '% ' prompt.

This is the essential pattern of all interaction with UNIX through the shell. A complete command is typed at the terminal, the shell executes the command and when this execution completes, it prompts for a new command. If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

An example of a useful command you can execute now is the *tset* command, which sets the default *erase* and *kill* characters on your terminal — the erase character erases the last character you typed and the kill character erases the entire line you have entered so far. By default, the erase character is '#' and the kill character is '@'. Most people who use CRT displays prefer to use the backspace (↑H) character as their erase character since it is then easier to see what you have typed so far. You can make this be true by typing

```
tset −e
```

which tells the program *tset* to set the erase character, and its default setting for this character is a backspace.

1.2. Flag arguments

A useful notion in UNIX is that of a *flag* argument. While many arguments to commands specify file names or user names some arguments rather specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the character '−' (hyphen). Thus the command

```
ls
```

will produce a list of the files in the current *working directory*. The option *−s* is the size option, and

```
ls −s
```

causes *ls* to also give, for each file the size of the file in blocks of 512 characters. The manual section for each command in the UNIX reference manual gives the available options for each command. The *ls* command has a large number of useful and interesting options. Most other commands have either no options or only one or two options. It is hard to remember options of commands which are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard to remember options.

1.3. Output to files

Commands that normally read input or write output on the terminal can also be executed with this input and/or output done to a file.

Thus suppose we wish to save the current date in a file called 'now'. The command

```
date
```

will print the current date on our terminal. This is because our terminal is the default *standard output* for the date command and the date command prints the date on its standard output. The shell lets us *redirect* the *standard output* of a command through a notation using the *metacharacter* '>' and the name of the file where output is to be placed. Thus the command

```
date > now
```

runs the *date* command such that its standard output is the file 'now' rather than the terminal. Thus this command places the current date and time into the file 'now'. It is important to know that the *date* command was unaware that its output was going to a file rather than to the terminal. The shell performed this *redirection* before the command began executing.

One other thing to note here is that the file 'now' need not have existed before the *date* command was executed; the shell would have created the file if it did not exist. And if the file did exist? If it had existed previously these previous contents would have been discarded! A shell option *noclobber* exists to prevent this from happening accidentally; it is discussed in section 2.2.

The system normally keeps files which you create with '>' and all other files. Thus the default is for files to be permanent. If you wish to create a file which will be removed automatically, you can begin its name with a '#' character, this 'scratch' character denotes the fact that the file will be a scratch file.* The system will remove such files after a couple of days, or

*Note that if your erase character is a '#', you will have to precede the '#' with a '\'. The fact that the '#' character is the old (pre-CRT) standard erase character means that it seldom appears in a file name, and allows this convention to be used for scratch files. If you are using a CRT, your erase character should be a ↑H, as we demonstrated in section 1.1 how this could be set up.

sooner if file space becomes very tight. Thus, in running the *date* command above, we don't really want to save the output forever, so we would more likely do

```
date > #now
```

1.4. Metacharacters in the shell

The shell has a large number of special characters (like '>') which indicate special functions. We say that these notations have *syntactic* and *semantic* meaning to the shell. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of *quotation* which allows us to use *metacharacters* without the shell treating them in any special way.

Metacharacters normally have effect only when the shell is reading our input. We need not worry about placing shell metacharacters in a letter we are sending via *mail*, or when we are typing in text or data to some other program. Note that the shell is only reading input when it has prompted with '% '.

1.5. Input from files; pipelines

We learned above how to *redirect* the *standard output* of a command to a file. It is also possible to redirect the *standard input* of a command from a file. This is not often necessary since most commands will read from a file whose name is given as an argument. We can give the command

```
sort < data
```

to run the *sort* command with standard input, where the command normally reads its input, from the file 'data'. We would more likely say

```
sort data
```

letting the *sort* command open the file 'data' for input itself since this is less to type.

We should note that if we just typed

```
sort
```

then the sort program would sort lines from its *standard input*. Since we did not *redirect* the standard input, it would sort lines as we typed them on the terminal until we typed a ↑D to indicate an end-of-file.

A most useful capability is the ability to combine the standard output of one command with the standard input of another, i.e. to run the commands in a sequence known as a *pipeline*. For instance the command

```
ls −s
```

normally produces a list of the files in our directory with the size of each in blocks of 512 characters. If we are interested in learning which of our files is largest we may wish to have this sorted by size rather than by name, which is the default way in which *ls* sorts. We could look at the many options of *ls* to see if there was an option to do this but would eventually discover that there is not. Instead we can use a couple of simple options of the *sort* command, combining it with *ls* to get what we want.

The *−n* option of sort specifies a numeric sort rather than an alphabetic sort. Thus

```
ls −s | sort −n
```

specifies that the output of the *ls* command run with the option *−s* is to be *piped* to the command *sort* run with the numeric sort option. This would give us a sorted list of our files by size, but with the smallest first. We could then use the *−r* reverse sort option and the *head* command in combination with the previous command doing

```
ls −s | sort −n −r | head −5
```

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the *sort* command asking it to sort numerically in reverse order (largest first). This output has then been run into the command *head* which gives us the first few lines. In this case we have asked *head* for the first 5 lines. Thus this command gives us the names and sizes of our 5 largest files.

The notation introduced above is called the *pipe* mechanism. Commands separated by '|' characters are connected together by the shell and the standard output of each is run into the standard input of the next. The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism; one important use of pipes which is illustrated there is in the routing of information to the line printer.

1.6. Filenames

Many commands to be executed will need the names of files as arguments. UNIX *pathnames* consist of a number of *components* separated by '/'. Each component except the last names a directory in which the next component resides, in effect specifying the *path* of directories to follow to reach the file. Thus the pathname

```
/etc/motd
```

specifies a file in the directory 'etc' which is a subdirectory of the *root* directory '/'. Within this directory the file named is 'motd' which stands for 'message of the day'. A *pathname* that begins with a slash is said to be an *absolute* pathname since it is specified from the absolute top of the entire directory hierarchy of the system (the *root*). *Pathnames* which do not begin with '/' are interpreted as starting in the current *working directory*, which is, by default, your *home* directory and can be changed dynamically by the *cd* change directory command. Such pathnames are said to be *relative* to the working directory since they are found by starting in the working directory and descending to lower levels of directories for each *component* of the pathname. If the pathname contains no slashes at all then the file is contained in the working directory itself and the pathname is merely the name of the file in this directory. Absolute pathnames have no relation to the working directory.

Most filenames consist of a number of alphanumeric characters and '.'s (periods). In fact, all printing characters except '/' (slash) may appear in filenames. It is inconvenient to have most non-alphabetic characters in filenames because many of these have special meaning to the shell. The character '.' (period) is not a shell-metacharacter and is often used to separate the *extension* of a file name from the base of the name. Thus

```
prog.c prog.o prog.errs prog.output
```

are four related files. They share a *base* portion of a name (a base portion being that part of the name that is left when a trailing '.' and following characters which are not '.' are stripped off). The file 'prog.c' might be the source for a C program, the file 'prog.o' the corresponding object file, the file 'prog.errs' the errors resulting from a compilation of the program and the file 'prog.output' the output of a run of the program.

If we wished to refer to all four of these files in a command, we could use the notation

```
prog.*
```

This word is expanded by the shell, before the command to which it is an argument is executed, into a list of names which begin with 'prog.'. The character '*' here matches any sequence (including the empty sequence) of characters in a file name. The names which match are alphabetically sorted and placed in the *argument list* of the command. Thus the command

```
echo prog.*
```

will echo the names

```
prog.c prog.errs prog.o prog.output
```

Note that the names are in sorted order here, and a different order than we listed them above. The *echo* command receives four words as arguments, even though we only typed one word as as argument directly. The four words were generated by *filename expansion* of the one input word.

Other notations for *filename expansion* are also available. The character '?' matches any single character in a filename. Thus

```
echo ? ?? ???
```

will echo a line of filenames; first those with one character names, then those with two character names, and finally those with three character names. The names of each length will be independently sorted.

Another mechanism consists of a sequence of characters between '[' and ']'. This metasequence matches any single character from the enclosed set. Thus

```
prog.[co]
```

will match

```
prog.c prog.o
```

in the example above. We can also place two characters around a '−' in this notation to denote a range. Thus

```
chap.[1−5]
```

might match files

```
chap.1 chap.2 chap.3 chap.4 chap.5
```

if they existed. This is shorthand for

```
chap.[12345]
```

and otherwise equivalent.

An important point to note is that if a list of argument words to a command (an *argument list)* contains filename expansion syntax, and if this filename expansion syntax fails to match any existing file names, then the shell considers this to be an error and prints a diagnostic

```
No match.
```

and does not execute the command.

Another very important point is that files with the character '.' at the beginning are treated specially. Neither '*' or '?' or the '[' ']' mechanism will match it. This prevents accidental matching of the filenames '.' and '..' in the working directory which have special meaning to the system, as well as other files such as *.cshrc* which are not normally visible. We will discuss the special role of the file *.cshrc* later.

Another filename expansion mechanism gives access to the pathname of the *home* directory of other users. This notation consists of the character '~' (tilde) followed by another users' login name. For instance the word '~bill' would map to the pathname '/usr/bill' if the home directory for 'bill' was '/usr/bill'. Since, on large systems, users may have login directories scattered over many different disk volumes with different prefix directory names, this notation provides a reliable way of accessing the files of other users.

A special case of this notation consists of a '~' alone, e.g. '~/mbox'. This notation is expanded by the shell into the file 'mbox' in your *home* directory, i.e. into '/usr/bill/mbox' for me on Ernie Co-vax, the UCB Computer Science Department VAX machine, where this document was prepared. This can be very useful if you have used *cd* to change to another directory and have found a file you wish to copy using *cp.* If I give the command

```
cp thatfile ~
```

the shell will expand this command to

```
cp thatfile /usr/bill
```

since my home directory is /usr/bill.

There also exists a mechanism using the characters '{' and '}' for abbreviating a set of words which have common parts but cannot be abbreviated by the above mechanisms because they are not files, are the names of files which do not yet exist, are not thus conveniently described. This mechanism will be described much later, in section 4.2, as it is used less frequently.

1.7. Quotation

We have already seen a number of metacharacters used by the shell. These metacharacters pose a problem in that we cannot use them directly as parts of words. Thus the command

```
echo *
```

will not echo the character '*'. It will either echo an sorted list of filenames in the current *working directory*, or print the message 'No match' if there are no files in the working directory.

The recommended mechanism for placing characters which are neither numbers, digits, '/', '.' or '−' in an argument word to a command is to enclose it with single quotation characters ''', i.e.

```
echo '*'
```

There is one special character '!' which is used by the *history* mechanism of the shell and which cannot be *escaped* by placing it within ''' characters. It and the character ''' itself can be preceded by a single '\' to prevent their special meaning. Thus

```
echo \'\!
```

prints

```
'!
```

These two mechanisms suffice to place any printing character into a word which is an argument to a shell command. They can be combined, as in

```
echo \''*'
```

which prints

```
'*
```

since the first '\' escaped the first ''' and the '*' was enclosed between ''' characters.

1.8. Terminating commands

When you are executing a command and the shell is waiting for it to complete there are several ways to force it to stop. For instance if you type the command

```
cat /etc/passwd
```

the system will print a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an INTERRUPT *signal* to the *cat* command by typing the DEL or RUBOUT key on your terminal.* Since *cat* does not take any precautions to avoid or otherwise handle this signal the INTERRUPT will cause it to terminate. The shell notices that *cat* has terminated and prompts you again with '% '. If you hit INTERRUPT

*Many users use *stty*(1) to change the interrupt character to ↑C.

again, the shell will just repeat its prompt since it handles INTERRUPT signals and chooses to continue to execute commands rather than terminating like *cat* did, which would have the effect of logging you out.

Another way in which many programs terminate is when they get an end-of-file from their standard input. Thus the *mail* program in the first example above was terminated when we typed a ↑D which generates an end-of-file from the standard input. The shell also terminates when it gets an end-of-file printing 'logout'; UNIX then logs you off the system. Since this means that typing too many ↑D's can accidentally log us off, the shell has a mechanism for preventing this. This *ignoreeof* option will be discussed in section 2.2.

If a command has its standard input redirected from a file, then it will normally terminate when it reaches the end of this file. Thus if we execute

```
mail bill < prepared.text
```

the mail command will terminate without our typing a ↑D. This is because it read to the end-of-file of our file 'prepared.text' in which we placed a message for 'bill' with an editor program. We could also have done

```
cat prepared.text | mail bill
```

since the *cat* command would then have written the text through the pipe to the standard input of the mail command. When the *cat* command completed it would have terminated, closing down the pipeline and the *mail* command would have received an end-of-file from it and terminated. Using a pipe here is more complicated than redirecting input so we would more likely use the first form. These commands could also have been stopped by sending an INTERRUPT.

Another possibility for stopping a command is to suspend its execution temporarily, with the possibility of continuing execution later. This is done by sending a STOP signal via typing a ↑Z. This signal causes all commands running on the terminal (usually one but more if a pipeline is executing) to become suspended. The shell notices that the command(s) have been suspended, types 'Stopped' and then prompts for a new command. The previously executing command has been suspended, but otherwise unaffected by the STOP signal. Any other commands can be executed while the original command remains suspended. The suspended command can be continued using the *fg* command with no arguments. The shell will then retype the command to remind you which command is being continued, and cause the command to resume execution. Unless any input files in use by the suspended command have been changed in the meantime, the suspension has no effect whatsoever on the execution of the command. This feature can be very useful during editing, when you need to look at another file before continuing. An example of command suspension follows.

```
% mail harold
Someone just copied a big file into my directory and its name is
↑Z
Stopped
% ls
funnyfile
prog.c
prog.o
% jobs
[1]  + Stopped              mail harold
% fg
mail harold
funnyfile. Do you know who did it?
EOT
%
```

In this example someone was sending a message to Harold and forgot the name of the file he wanted to mention. The mail command was suspended by typing ↑Z. When the shell noticed that the mail program was suspended, it typed 'Stopped' and prompted for a new command. Then the *ls* command was typed to find out the name of the file. The *jobs* command was run to find out which command was suspended. At this time the *fg* command was typed to continue execution of the mail program. Input to the mail program was then continued and ended with a ↑D which indicated the end of the message at which time the mail program typed EOT. The *jobs* command will show which commands are suspended. The ↑Z should only be typed at the beginning of a line since everything typed on the current line is discarded when a signal is sent from the keyboard. This also happens on INTERRUPT, and QUIT signals. More information on suspending jobs and controlling them is given in section 2.6.

If you write or run programs which are not fully debugged then it may be necessary to stop them somewhat ungracefully. This can be done by sending them a QUIT signal, sent by typing a ↑\. This will usually provoke the shell to produce a message like:

```
Quit (Core dumped)
```

indicating that a file 'core' has been created containing information about the program 'a.out's state when it terminated due to the QUIT signal. You can examine this file yourself, or forward information to the maintainer of the program telling him/her where the *core file* is.

If you run background commands (as explained in section 2.6) then these commands will ignore INTERRUPT and QUIT signals at the terminal. To stop them you must use the *kill* command. See section 2.6 for an example.

If you want to examine the output of a command without having it move off the screen as the output of the

```
cat /etc/passwd
```

command will, you can use the command

```
more /etc/passwd
```

The *more* program pauses after each complete screenful and types '−−More−−' at which point you can hit a space to get another screenful, a return to get another line, or a 'q' to end the *more* program. You can also use more as a filter, i.e.

```
cat /etc/passwd | more
```

works just like the more simple more command above.

For stopping output of commands not involving *more* you can use the ↑S key to stop the typeout. The typeout will resume when you hit ↑Q or any other key, but ↑Q is normally used because it only restarts the output and does not become input to the program which is running. This works well on low-speed terminals, but at 9600 baud it is hard to type ↑S and ↑Q fast enough to paginate the output nicely, and a program like *more* is usually used.

An additional possibility is to use the ↑O flush output character; when this character is typed, all output from the current command is thrown away (quickly) until the next input read occurs or until the next shell prompt. This can be used to allow a command to complete without having to suffer through the output on a slow terminal; ↑O is a toggle, so flushing can be turned off by typing ↑O again while output is being flushed.

1.9. What now?

We have so far seen a number of mechanisms of the shell and learned a lot about the way in which it operates. The remaining sections will go yet further into the internals of the shell, but you will surely want to try using the shell before you go any further. To try it you can log in to UNIX and type the following command to the system:

```
chsh myname /bin/csh
```

Here 'myname' should be replaced by the name you typed to the system prompt of 'login:' to get onto the system. Thus I would use 'chsh bill /bin/csh'. **You only have to do this once; it**

takes effect at next login. You are now ready to try using *csh.*

Before you do the 'chsh' command, the shell you are using when you log into the system is '/bin/sh'. In fact, much of the above discussion is applicable to '/bin/sh'. The next section will introduce many features particular to *csh* so you should change your shell to *csh* before you begin reading it.

2. Details on the shell for terminal users

2.1. Shell startup and termination

When you login, the shell is started by the system in your *home* directory and begins by reading commands from a file *.cshrc* in this directory. All shells which you may start during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A *login shell*, executed after you login to the system, will, after it reads commands from *.cshrc*, read commands from a file *.login* also in your home directory. This file contains commands which you wish to do each time you login to the UNIX system. My *.login* file looks something like:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
echo "${prompt}users" ; users
alias ts \
        'set noglob ; eval `tset −s −m dialup:c100rv4pna −m plugboard:?hp2621nl *`';
ts; stty intr ↑C kill ↑U crt
set time=15 history=10
msgs −f
if (−e $mail) then
        echo "${prompt}mail"
        mail
endif
```

This file contains several commands to be executed by UNIX each time I login. The first is a *set* command which is interpreted directly by the shell. It sets the shell variable *ignoreeof* which causes the shell to not log me off if I hit ↑D. Rather, I use the *logout* command to log off of the system. By setting the *mail* variable, I ask the shell to watch for incoming mail to me. Every 5 minutes the shell looks for this file and tells me if more mail has arrived there. An alternative to this is to put the command

```
biff y
```

in place of this *set;* this will cause me to be notified immediately when mail arrives, and to be shown the first few lines of the new message.

Next I set the shell variable 'time' to '15' causing the shell to automatically print out statistics lines for commands which execute for at least 15 seconds of CPU time. The variable 'history' is set to 10 indicating that I want the shell to remember the last 10 commands I type in its *history list*, (described later).

I create an *alias* "ts" which executes a *tset*(1) command setting up the modes of the terminal. The parameters to *tset* indicate the kinds of terminal which I usually use when not on a hardwired port. I then execute "ts" and also use the *stty* command to change the interrupt character to ↑C and the line kill character to ↑U.

I then run the 'msgs' program, which provides me with any system messages which I have not seen before; the '−f' option here prevents it from telling me anything if there are no new messages. Finally, if my mailbox file exists, then I run the 'mail' program to process my mail.

When the 'mail' and 'msgs' programs finish, the shell will finish processing my *.login* file and begin reading commands from the terminal, prompting for each with '% '. When I log off (by giving the *logout* command) the shell will print 'logout' and execute commands from the file '.logout' if it exists in my home directory. After that the shell will terminate and UNIX will log me off the system. If the system is not going down, I will receive a new login message. In any case, after the 'logout' message the shell is committed to terminating and will take no further input from my terminal.

2.2. Shell variables

The shell maintains a set of *variables.* We saw above the variables *history* and *time* which had values '10' and '15'. In fact, each shell variable has as value an array of zero or more *strings.* Shell variables may be assigned values by the set command. It has several forms, the most useful of which was given above and is

```
set name=value
```

Shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable *path.* This variable contains a sequence of directory names where the shell searches for commands. The *set* command with no arguments shows the value of all variables currently defined (we usually say *set)* in the shell. The default value for path will be shown by *set* to be

```
% set
argv      ()
cwd       /usr/bill
home      /usr/bill
path      (. /usr/ucb /bin /usr/bin)
prompt    %
shell     /bin/csh
status    0
term      c100rv4pna
user      bill
%
```

This output indicates that the variable path points to the current directory '.' and then '/usr/ucb', '/bin' and '/usr/bin'. Commands which you may write might be in '.' (usually one of your directories). Commands developed at Berkeley, live in '/usr/ucb' while commands developed at Bell Laboratories live in '/bin' and '/usr/bin'.

A number of locally developed programs on the system live in the directory '/usr/local'. If we wish that all shells which we invoke to have access to these new programs we can place the command

```
set path=(. /usr/ucb /bin /usr/bin /usr/local)
```

in our file *.cshrc* in our home directory. Try doing this and then logging out and back in and do

```
set
```

again to see that the value assigned to *path* has changed.

One thing you should be aware of is that the shell examines each directory which you insert into your path and determines which commands are contained there. Except for the current directory '.', which the shell treats specially, this means that if commands are added to a directory in your search path after you have started the shell, they will not necessarily be found by the shell. If you wish to use a command which has been added in this way, you should give the command

```
rehash
```

to the shell, which will cause it to recompute its internal table of command locations, so that it will find the newly added command. Since the shell has to look in the current directory '.' on each command, placing it at the end of the path specification usually works equivalently and reduces overhead.

Other useful built in variables are the variable *home* which shows your home directory, *cwd* which contains your current working directory, the variable *ignoreeof* which can be set in your *.login* file to tell the shell not to exit when it receives an end-of-file from a terminal (as described above). The variable 'ignoreeof' is one of several variables which the shell does not care about the value of, only whether they are *set* or *unset.* Thus to set this variable you simply do

```
set ignoreeof
```

and to unset it do

```
unset ignoreeof
```

These give the variable 'ignoreeof' no value, but none is desired or required.

Finally, some other built-in shell variables of use are the variables *noclobber* and *mail.* The metasyntax

```
> filename
```

which redirects the standard output of a command will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

```
set noclobber
```

in your *.login* file. Then trying to do

```
date > now
```

would cause a diagnostic if 'now' existed already. You could type

```
date >! now
```

if you really wanted to overwrite the contents of 'now'. The '>!' is a special metasyntax indicating that clobbering the file is ok.†

2.3. The shell's history list

The shell can maintain a *history list* into which it places the words of previous commands. It is possible to use a notation to reuse commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the shell. In this example we have a very simple C program which has a bug (or two) in it in the file 'bug.c', which we 'cat' out on our terminal. We then try to run the C compiler on it, referring to the file again as '!$', meaning the last argument to the previous command. Here the '!' is the history mechanism invocation metacharacter, and the '$' stands for the last argument, by analogy to '$' in the editor which stands for the end of the line. The shell echoed the command, as it would have been typed without use of the history mechanism, and then executed it. The compilation yielded error diagnostics so we now run the editor on the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as '!c', which repeats the last command which started with the letter 'c'. If there were other commands starting with 'c' done recently we could have said '!cc' or even '!cc:p' which would have printed the last command starting with 'cc' without executing it.

†The space between the '!' and the word 'now' is critical here, as '!now' would be an invocation of the *history* mechanism, and have a totally different effect.

```
% cat bug.c
main()

{
        printf("hello);
}
% cc !$
cc bug.c
"bug.c", line 4: newline in string or char constant
"bug.c", line 5: syntax error
% ed !$
ed bug.c
29
4s/);/"&/p
        printf("hello");
w
30
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
30
4s/lo/lo\\n/p
        printf("hello\n");
w
32
q
% !c −o bug
cc bug.c −o bug
% size a.out bug
a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b
% ls −l !*
ls −l a.out bug
−rwxr−xr−x 1 bill      3932 Dec 19 09:41 a.out
−rwxr−xr−x 1 bill      3932 Dec 19 09:42 bug
% bug
hello
% num bug.c | spp
spp: Command not found.
% ↑spp↑ssp
num bug.c | ssp
   1  main()
   3  {
   4          printf("hello\n");
   5  }
% !! | lpr
num bug.c | ssp | lpr
%
```

After this recompilation, we ran the resulting 'a.out' file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra '−o bug' telling the compiler to place the resultant binary in the file 'bug' rather than 'a.out'. In general, the history mechanisms may be used anywhere in the formation of new commands and other characters may be placed before and after the substituted commands.

We then ran the 'size' command to see how large the binary program images we have created were, and then an 'ls −l' command with the same argument list, denoting the argument list '*'. Finally we ran the program 'bug' to see that its output is indeed correct.

To make a numbered listing of the program we ran the 'num' command on the file 'bug.c'. In order to compress out blank lines in the output of 'num' we ran the output through the filter 'ssp', but misspelled it as spp. To correct this we used a shell substitute, placing the old text and new text between '↑' characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with '!!', but sent its output to the line printer.

There are other mechanisms available for repeating commands. The *history* command prints out a number of previous commands with numbers by which they can be referenced. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in the C shell manual pages in the UNIX Programmers Manual.

2.4. Aliases

The shell has an *alias* mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using shell command files, but these take place in another instance of the shell and cannot directly affect the current shells environment or involve commands such as *cd* which must be done in the current shell.

As an example, suppose that there is a new version of the mail program on the system called 'newmail' you wish to use, rather than the standard mail program which is called 'mail'. If you place the shell command

```
alias mail newmail
```

in your *.cshrc* file, the shell will transform an input line of the form

```
mail bill
```

into a call on 'newmail'. More generally, suppose we wish the command 'ls' to always show sizes of files, that is to always do '−s'. We can do

```
alias ls ls −s
```

or even

```
alias dir ls −s
```

creating a new command syntax 'dir' which does an 'ls −s'. If we say

```
dir ~bill
```

then the shell will translate this to

```
ls −s /mnt/bill
```

Thus the *alias* mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

```
alias cd 'cd \!* ; ls '
```

would do an *ls* command after each change directory *cd* command. We enclosed the entire alias definition in ''' characters to prevent most substitutions from occurring and the character ';' from being recognized as a metacharacter. The '!' here is escaped with a '\' to prevent it from being interpreted when the alias command is typed in. The '\!*' here substitutes the entire argument list to the pre-aliasing *cd* command, without giving an error if there were no arguments. The ';' separating commands is used here to indicate that one command is to be done and then the next. Similarly the definition

```
alias whois 'grep \!↑ /etc/passwd'
```

defines a command which looks up its first argument in the password file.

Warning: The shell currently reads the *.cshrc* file each time it starts up. If you place a large number of commands there, shells will tend to start slowly. A mechanism for saving the shell environment after reading the *.cshrc* file and quickly restoring it is under development, but for now you should try to limit the number of aliases you have to a reasonable number... 10 or 15 is reasonable, 50 or 60 will cause a noticeable delay in starting up shells, and make the system seem sluggish when you execute commands from within the editor and other programs.

2.5. More redirection; >> and >&

There are a few more notations useful to the terminal user which have not been introduced yet.

In addition to the standard output, commands also have a *diagnostic output* which is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can do

```
command >& file
```

The '>&' here tells the shell to route both the diagnostic output and the standard output into 'file'. Similarly you can give the command

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the line printer daemon *lpr*.#

Finally, it is possible to use the form

```
command >> file
```

to place output at the end of an existing file.†

#A command form

```
command >&! file
```

exists, and is used when *noclobber* is set and *file* already exists.

†If *noclobber* is set, then an error will result if *file* does not exist, otherwise the shell will create *file* if it doesn't exist. A form

```
command >>! file
```

makes it not be an error for file to not exist when *noclobber* is set.

2.6. Jobs; Background, Foreground, or Suspended

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single *job* is created by the shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the shell creates a job. Some lines that create jobs (one per line) are

```
sort < data
ls −s | sort −n | head −5
mail harold
```

If the metacharacter '&' is typed at the end of the commands, then the job is started as a *background* job. This means that the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs *in the background* at the same time that normal jobs, called *foreground* jobs, continue to be read and executed by the shell one at a time. Thus

```
du > usage &
```

would run the *du* program, which reports on the disk usage of your working directory (as well as any directories below it), put the output into the file 'usage' and return immediately with a prompt for the next command without out waiting for *du* to finish. The *du* program would continue executing in the background until it finished, even though you can type and execute more commands in the mean time. When a background job terminates, a message is typed by the shell just before the next prompt telling you that the job has completed. In the following example the *du* job finishes sometime during the execution of the *mail* command and its completion is reported just before the prompt after the *mail* job is finished.

```
% du > usage &
[1] 503
% mail bill
How do you know when a background job is finished?
EOT
[1] − Done                    du > usage
%
```

If the job did not terminate normally the 'Done' message might say something else like 'Killed'. If you want the terminations of background jobs to be reported at the time they occur (possibly interrupting the output of other foreground jobs), you can set the *notify* variable. In the previous example this would mean that the 'Done' message might have come right in the middle of the message to Bill. Background jobs are unaffected by any signals from the keyboard like the STOP, INTERRUPT, or QUIT signals mentioned earlier.

Jobs are recorded in a table inside the shell until they terminate. In this table, the shell remembers the command names, arguments and the *process numbers* of all commands in the job as well as the working directory where the job was started. Each job in the table is either running *in the foreground* with the shell waiting for it to terminate, running *in the background*, or *suspended*. Only one job can be running in the foreground at one time, but several jobs can be suspended or running in the background at once. As each job is started, it is assigned a small identifying number called the *job number* which can be used later to refer to the job in the commands described below. Job numbers remain the same until the job terminates and then are re-used.

When a job is started in the backgound using '&', its number, as well as the process numbers of all its (top level) commands, is typed by the shell before prompting you for another command. For example,

```
% ls −s | sort −n > usage &
[2] 2034 2035
%
```

runs the 'ls' program with the '−s' options, pipes this output into the 'sort' program with the '−n' option which puts its output into the file 'usage'. Since the '&' was at the end of the line, these two programs were started together as a background job. After starting the job, the shell prints the job number in brackets (2 in this case) followed by the process number of each program started in the job. Then the shell immediates prompts for a new command, leaving the job running simultaneously.

As mentioned in section 1.8, foreground jobs become *suspended* by typing ↑Z which sends a STOP signal to the currently running foreground job. A background job can become suspended by using the *stop* command described below. When jobs are suspended they merely stop any further progress until started again, either in the foreground or the backgound. The shell notices when a job becomes stopped and reports this fact, much like it reports the termination of background jobs. For foreground jobs this looks like

```
% du > usage
↑Z
Stopped
%
```

'Stopped' message is typed by the shell when it notices that the *du* program stopped. For background jobs, using the *stop* command, it is

```
% sort usage &
[1] 2345
% stop %1
[1] + Stopped (signal)      sort usage
%
```

Suspending foreground jobs can be very useful when you need to temporarily change what you are doing (execute other commands) and then return to the suspended job. Also, foreground jobs can be suspended and then continued as background jobs using the *bg* command, allowing you to continue other work and stop waiting for the foreground job to finish. Thus

```
% du > usage
↑Z
Stopped
% bg
[1] du > usage &
%
```

starts 'du' in the foreground, stops it before it finishes, then continues it in the background allowing more foreground commands to be executed. This is especially helpful when a foreground job ends up taking longer than you expected and you wish you had started it in the backgound in the beginning.

All *job control* commands can take an argument that identifies a particular job. All job name arguments begin with the character '%', since some of the job control commands also accept process numbers (printed by the *ps* command.) The default job (when no argument is given) is called the *current* job and is identified by a '+' in the output of the *jobs* command, which shows you which jobs you have. When only one job is stopped or running in the background (the usual case) it is always the current job thus no argument is needed. If a job is stopped while running in the foreground it becomes the *current* job and the existing current job becomes the *previous* job − identified by a '−' in the output of *jobs*. When the current job terminates, the previous job becomes the current job. When given, the argument is either '%−' (indicating the previous job); '%#', where # is the job number; '%pref' where pref is some unique prefix of the command name and arguments of one of the jobs; or '%?' followed by some string found in only one of the jobs.

The *jobs* command types the table of jobs, giving the job number, commands and status ('Stopped' or 'Running') of each backgound or suspended job. With the '−l' option the process numbers are also typed.

```
% du > usage &
[1] 3398
% ls −s | sort −n > myfile &
[2] 3405
% mail bill
↑Z
Stopped
% jobs
[1] − Running           du > usage
[2]   Running           ls −s | sort −n > myfile
[3] + Stopped           mail bill
% fg %ls
ls −s | sort −n > myfile
% more myfile
```

The *fg* command runs a suspended or background job in the foreground. It is used to restart a previously suspended job or change a background job to run in the foreground (allowing signals or input from the terminal). In the above example we used *fg* to change the 'ls' job from the background to the foreground since we wanted to wait for it to finish before looking at its output file. The *bg* command runs a suspended job in the background. It is usually used after stopping the currently running foreground job with the STOP signal. The combination of the STOP signal and the *bg* command changes a foreground job into a background job. The *stop* command suspends a background job.

The *kill* command terminates a background or suspended job immediately. In addition to jobs, it may be given process numbers as arguments, as printed by *ps*. Thus, in the example above, the running *du* command could have been terminated by the command

```
% kill %1
[1] Terminated          du > usage
%
```

The *notify* command (not the variable mentioned earlier) indicates that the termination of a specific job should be reported at the time it finishes instead of waiting for the next prompt.

If a job running in the background tries to read input from the terminal it is automatically stopped. When such a job is then run in the foreground, input can be given to the job. If desired, the job can be run in the background again until it requests input again. This is illustrated in the following sequence where the 's' command in the text editor might take a long time.

```
% ed bigfile
120000
1,$s/thisword/thatword/
↑Z
Stopped
% bg
[1] ed bigfile &
%
 . . . some foreground commands
[1] Stopped (tty input)     ed bigfile
% fg
```

```
ed bigfile
w
120000
q
%
```

So after the 's' command was issued, the 'ed' job was stopped with ↑Z and then put in the background using *bg*. Some time later when the 's' command was finished, *ed* tried to read another command and was stopped because jobs in the backgound cannot read from the terminal. The *fg* command returned the 'ed' job to the foreground where it could once again accept commands from the terminal.

The command

```
stty tostop
```

causes all background jobs run on your terminal to stop when they are about to write output to the terminal. This prevents messages from background jobs from interrupting foreground job output and allows you to run a job in the background without losing terminal output. It also can be used for interactive programs that sometimes have long periods without interaction. Thus each time it outputs a prompt for more input it will stop before the prompt. It can then be run in the foreground using *fg*, more input can be given and, if necessary stopped and returned to the background. This *stty* command might be a good thing to put in your *.login* file if you do not like output from background jobs interrupting your work. It also can reduce the need for redirecting the output of background jobs if the output is not very big:

```
% stty tostop
% wc hugefile &
[1] 10387
% ed text
. . . some time later
q
[1] Stopped (tty output)      wc hugefile
% fg wc
wc hugefile
  13371   30123   302577
% stty −tostop
```

Thus after some time the 'wc' command, which counts the lines, words and characters in a file, had one line of output. When it tried to write this to the terminal it stopped. By restarting it in the foreground we allowed it to write on the terminal exactly when we were ready to look at its output. Programs which attempt to change the mode of the terminal will also block, whether or not *tostop* is set, when they are not in the foreground, as it would be very unpleasant to have a background job change the state of the terminal.

Since the *jobs* command only prints jobs started in the currently executing shell, it knows nothing about background jobs started in other login sessions or within shell files. The *ps* can be used in this case to find out about background jobs not started in the current shell.

2.7. Working Directories

As mentioned in section 1.6, the shell is always in a particular *working directory*. The 'change directory' command *chdir* (its short form *cd* may also be used) changes the working directory of the shell, that is, changes the directory you are located in.

It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. The 'make directory' command, *mkdir*, creates a new directory. The *pwd* ('print working directory') command reports the absolute pathname of the working directory of the shell, that is, the directory you are located in. Thus in the example below:

```
% pwd
/usr/bill
% mkdir newpaper
% chdir newpaper
% pwd
/usr/bill/newpaper
%
```

the user has created and moved to the directory *newpaper*. where, for example, he might place a group of related files.

No matter where you have moved to in a directory hierarchy, you can return to your 'home' login directory by doing just

```
cd
```

with no arguments. The name '..' always means the directory above the current one in the hierarchy, thus

```
cd ..
```

changes the shell's working directory to the one directly above the current one. The name '..' can be used in any pathname, thus,

```
cd ../programs
```

means change to the directory 'programs' contained in the directory above the current one. If you have several directories for different projects under, say, your home directory, this shorthand notation permits you to switch easily between them.

The shell always remembers the pathname of its current working directory in the variable *cwd*. The shell can also be requested to remember the previous directory when you change to a new working directory. If the 'push directory' command *pushd* is used in place of the *cd* command, the shell saves the name of the current working directory on a *directory stack* before changing to the new one. You can see this list at any time by typing the 'directories' command *dirs*.

```
% pushd newpaper/references
~/newpaper/references ~
% pushd /usr/lib/tmac
/usr/lib/tmac ~/newpaper/references ~
% dirs
/usr/lib/tmac ~/newpaper/references ~
% popd
~/newpaper/references ~
% popd
~

%
```

The list is printed in a horizontal line, reading left to right, with a tilde (~) as shorthand for your home directory—in this case '/usr/bill'. The directory stack is printed whenever there is more than one entry on it and it changes. It is also printed by a *dirs* command. *Dirs* is usually faster and more informative than *pwd* since it shows the current working directory as well as any other directories remembered in the stack.

The *pushd* command with no argument alternates the current directory with the first directory in the list. The 'pop directory' *popd* command without an argument returns you to the directory you were in prior to the current one, discarding the previous current directory from the stack (forgetting it). Typing *popd* several times in a series takes you backward through the directories you had been in (changed to) by *pushd* command. There are other options to *pushd* and *popd* to manipulate the contents of the directory stack and to change to directories not at the top of the stack; see the *csh* manual page for details.

Since the shell remembers the working directory in which each job was started, it warns you when you might be confused by restarting a job in the foreground which has a different working directory than the current working directory of the shell. Thus if you start a background job, then change the shell's working directory and then cause the background job to run in the foreground, the shell warns you that the working directory of the currently running foreground job is different from that of the shell.

```
% dirs −l
/mnt/bill
% cd myproject
% dirs
~/myproject
% ed prog.c
1143
↑Z
Stopped
% cd ..
% ls
myproject
textfile
% fg
ed prog.c (wd: ~/myproject)
```

This way the shell warns you when there is an implied change of working directory, even though no cd command was issued. In the above example the 'ed' job was still in '/mnt/bill/project' even though the shell had changed to '/mnt/bill'. A similar warning is given when such a foreground job terminates or is suspended (using the STOP signal) since the return to the shell again implies a change of working directory.

```
% fg
ed prog.c (wd: ~/myproject)
 . . . after some editing
q
(wd now: ~)
%
```

These messages are sometimes confusing if you use programs that change their own working directories, since the shell only remembers which directory a job is started in, and assumes it stays there. The '−l' option of *jobs* will type the working directory of suspended or background jobs when it is different from the current working directory of the shell.

2.8. Useful built-in commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The *alias* command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given only one argument such as

```
alias ls
```

to show the current alias for, e.g., 'ls'.

The *echo* command prints its arguments. It is often used in *shell scripts* or as an interactive command to see what filename expansions will produce.

The *history* command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called *prompt*.

By placing a '!' character in its value the shell will there substitute the number of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

```
set prompt='\! % '
```

Note that the '!' character had to be *escaped* here even within '' characters.

The *limit* command is used to restrict use of resources. With no arguments it prints the current limitations:

```
cputime         unlimited
filesize        unlimited
datasize        5616 kbytes
stacksize       512 kbytes
coredumpsize    unlimited
```

Limits can be set, e.g.:

```
limit coredumpsize 128k
```

Most reasonable units abbreviations will work; see the *csh* manual page for more details.

The *logout* command can be used to terminate a login shell which has *ignoreeof* set.

The *rehash* command causes the shell to recompute a table of where commands are located. This is necessary if you add a command to a directory in the current shell's search path and wish the shell to find it, since otherwise the hashing algorithm may tell the shell that the command wasn't in that directory when the hash table was computed.

The *repeat* command can be used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could do

```
repeat 5 cat one >> five
```

The *setenv* command can be used to set variables in the environment. Thus

```
setenv TERM adm3a
```

will set the value of the environment variable TERM to 'adm3a'. A user program *printenv* exists which will print out the environment. It might then show:

```
% printenv
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
%
```

The *source* command can be used to force the current shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the *.cshrc* file which you wish to take effect before the next time you login.

The *time* command can be used to cause a command to be timed no matter how much CPU time it takes. Thus

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8% 2+1k 3+2io 1pf+0w
% time wc /etc/rc /usr/bill/rc
    52    178   1347 /etc/rc
    52    178   1347 /usr/bill/rc
   104    356   2694 total
0.1u 0.1s 0:00 13% 3+3k 5+3io 7pf+0w
%
```

indicates that the *cp* command used a negligible amount of user time (u) and about 1/10th of a system time (s); the elapsed time was 1 second (0:01), there was an average memory usage of 2k bytes of program space and 1k bytes of data space over the cpu time involved (2+1k); the program did three disk reads and two disk writes (3+2io), and took one page fault and was not swapped (1pf+0w). The word count command *wc* on the other hand used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage '13%' indicates that over the period when it was active the command 'wc' used an average of 13 percent of the available CPU cycles of the machine.

The *unalias* and *unset* commands can be used to remove aliases and variable definitions from the shell, and *unsetenv* removes variables from the environment.

2.9. What else?

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the *foreach* built-in command which can be used to run the same command sequence with a number of different arguments.

If you intend to use UNIX a lot you you should look through the rest of this document and the shell manual pages to become familiar with the other facilities which are available to you.

3. Shell control structures and command scripts

3.1. Introduction

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called *shell scripts.* We here detail those features of the shell useful to the writers of such scripts.

3.2. Make

It is important to first note what shell scripts are *not* useful for. There is a program called *make* which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a *makefile* which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this *makefile.* This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a *makefile* may be created which defines how different versions of the document are to be created and which options of *nroff* or *troff* are appropriate.

3.3. Invocation and the argv variable

A *csh* command script may be interpreted by saying

```
% csh script ...
```

where *script* is the name of the file containing a group of *csh* commands and '...' is replaced by a sequence of arguments. The shell places these arguments in the variable *argv* and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file 'script' executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a '#' character) then a '/bin/csh' will automatically be invoked to execute 'script' when you type

```
script
```

If the file does not begin with a '#' then the standard shell '/bin/sh' will be used to execute it. This allows you to convert your older shell scripts to use *csh* at your convenience.

3.4. Variable substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as *variable substitution* is done on these words. Keyed by the character '$' this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable *argv* to be echoed to the output of the shell script. It is an error for *argv* to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to '1' if name is *set* or to '0' if name is not *set.* It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable *name*. Thus

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable which has several values. Thus

```
$argv[1]
```

gives the first component of *argv* or in the example above 'a'. Similarly

```
$argv[$#argv]
```

would give 'c', and

```
$argv[1-2]
```

would give 'a b'. Other notations useful in shell scripts are

$*n*

where *n* is an integer as a shorthand for

$argv[*n*]

the *n th* parameter and

```
$*
```

which is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names. The form

```
$<
```

is quite special and is replaced by the next line of input read from the shell's standard input (not the script it is reading). This is useful for writing shell scripts that are interactive, reading commands from the terminal, or even writing a shell script that acts as a filter, reading lines from its input file. Thus the sequence

```
echo 'yes or no?\c'
set a=($<)
```

would write out the prompt 'yes or no?' without a newline and then read the answer into the variable 'a'. In this case '$#a' would be '0' if either a blank line or end-of-file (↑D) was typed.

One minor difference between '$*n*' and '$argv[*n*]' should be noted here. The form '$argv[*n*]' will yield an error if *n* is not in the range '1−$#argv' while '$n' will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form 'n−': if there are less than *n* components of the given variable then no words are substituted. A range of the form 'm−n' likewise returns an empty vector without giving an error when *m* exceeds the number of elements of the given variable, provided the subscript *n* is in range.

3.5. Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations '==' and '!=' compare strings and the operators '&&' and '||' implement the boolean and/or operations. The special operators '=~' and '!~' are similar to '==' and '!=' except that the string on the right side can have pattern matching characters (like *, ? or []) and the test is whether the string on the left matches the pattern on the right.

The shell also allows file enquiries of the form

−? filename

where '?' is replace by a number of single characters. For instance the expression primitive

−e filename

tell whether the file 'filename' exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form '{ command }' which returns true, i.e. '1' if the command succeeds exiting normally with exit status 0, or '0' if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable '$status' examined in the next command. Since '$status' is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see the manual section for the shell.

3.6. Sample shell script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i !~ *.c) continue  # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script makes use of the *foreach* command, which causes the shell to execute the commands between the *foreach* and the matching *end* for each of the values given between '(' and ')' with the named variable, in this case 'i' set to successive values in the list. Within this loop we may use the command *break* to stop executing the loop and *continue* to prematurely terminate one iteration and begin the next. After the *foreach* loop the iteration variable (*i* in this case) has the value at the last iteration.

We set the variable *noglob* here to prevent filename expansion of the members of *argv*. This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a '$' variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form

```
if ( expression ) then
        command
        ...
endif
```

The placement of the keywords here is **not** flexible due to the current implementation of the shell.†

†The following two formats are not currently acceptable to the shell:

```
if ( expression )               # Won't work!
then
        command
        ...
endif
```

and

```
if ( expression ) then command endif          # Won't work
```

The shell does have another form of the if statement of the form

```
if ( expression ) command
```

which can be written

```
if ( expression ) \
        command
```

Here we have escaped the newline for the sake of appearance. The command must not involve '|', '&' or ';' and must not be another control command. The second form requires the final '\' to **immediately** precede the end-of-line.

The more general *if* statements above also admit a sequence of *else*—*if* pairs followed by a single *else* and an *endif*, e.g.:

```
if ( expression ) then
        commands
else if (expression ) then
        commands
...

else
        commands
endif
```

Another important mechanism used in shell scripts is the ':' modifier. We can use the modifier ':r' here to extract a root of a filename or ':e' to extract the *extension*. Thus if the variable *i* has the value '/mnt/foo.bar' then

```
% echo $i $i:r $i:e
/mnt/foo.bar /mnt/foo bar
%
```

shows how the ':r' modifier strips off the trailing '.bar' and the the ':e' modifier leaves only the 'bar'. Other modifiers will take off the last component of a pathname leaving the head ':h' or all but the last component of a pathname leaving the tail ':t'. These modifiers are fully described in the *csh* manual pages in the programmers manual. It is also possible to use the *command substitution* mechanism described in the next major section to perform modifications on strings to then reenter the shells environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the ':' modification mechanism.# Finally, we note that the character '#' lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a '#' are discarded by the shell. This character can be quoted using ''' or '\' to place it in an argument word.

#It is also important to note that the current implementation of the shell limits the number of ':' modifiers on a '$' substitution to 1. Thus

```
% echo $i $i:h:t
/a/b/c /a/b:t
%
```

does not do what one would expect.

3.7. Other control structures

The shell also has control structures *while* and *switch* similar to those of C. These take the forms

```
while ( expression )
        commands
end
```

and

```
switch ( word )

case str1:
        commands
        breaksw

...

case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

For details see the manual section for *csh*. C programmers should note that we use *breaksw* to exit from a *switch* while *break* exits a *while* or *foreach* loop. A common mistake to make in *csh* scripts is to use *break* rather than *breaksw* in switches.

Finally, *csh* allows a *goto* statement, with labels looking like they do in C, i.e.:

```
loop:
        commands
        goto loop
```

3.8. Supplying input to commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This is different from previous shells running under UNIX. It allows shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file

```
% cat deblank
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/↑[ ]*//
w
q
'EOF'
end
%
```

The notation '<< 'EOF'' means that the standard input for the *ed* command is to come from the text in the shell script file up to the next line consisting of exactly ''EOF''. The fact that the 'EOF' is enclosed in ''' characters, i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the '<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form '1,$' in our editor script we needed to insure that this '$' was not variable substituted. We could also have insured this by preceding the '$' here with a '\', i.e.:

```
1,\$s/↑[ ]*//
```

but quoting the 'EOF' terminator is a more reliable way of achieving the same thing.

3.9. Catching interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where *label* is a label in our program. If an interrupt is received the shell will do a 'goto label' and we can remove the temporary files and then do an *exit* command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status '1'.

3.10. What else?

There are other features of the shell useful to writers of shell procedures. The *verbose* and *echo* options and the related *−v* and *−x* command line options can be used to help trace the actions of the shell. The *−n* option causes the shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that *csh* will not execute shell scripts which do not begin with the character '#', that is shell scripts that do not begin with a comment. Similarly, the '/bin/sh' on your system may well defer to 'csh' to interpret shell scripts which begin with '#'. This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using '"' which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as ''' does.

4. Other, less commonly used, shell features

4.1. Loops at the terminal; variables as vectors

It is occasionally useful to use the *foreach* control structure at the terminal to aid in performing a number of similar commands. For instance, there were at one point three shells in use on the Cory UNIX system at Cory Hall, '/bin/sh', '/bin/nsh', and '/bin/csh'. To count the number of persons using each shell one could have issued the commands

```
% grep −c cshS /etc/passwd
27
% grep −c nshS /etc/passwd
128
% grep −c −v shS /etc/passwd
430
%
```

Since these commands are very similar we can use *foreach* to do this more easily.

```
% foreach i ('shS' 'cshS' '−v shS')
? grep −c Si /etc/passwd
? end
27
128
430
%
```

Note here that the shell prompts for input with '? ' when reading the body of the loop.

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

```
% set a=(`ls`)
% echo Sa
csh.n csh.rm
% ls
csh.n
csh.rm
% echo S#a
2
%
```

The *set* command here gave the variable *a* a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within '`' characters is converted by the shell to a list of words. You can also place the '`' quoted string within '"' characters to take each (non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier ':x' exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

4.2. Braces { ... } in argument expansion

Another form of filename expansion, alluded to before involves the characters '{' and '}'. These characters specify that the contained strings, separated by ',' are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

```
mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories 'hdrs', 'retrofit' and 'csh' in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

4.3. Command substitution

A command enclosed in '`' characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to do

```
set pwd=`pwd`
```

to save the current directory in the variable *pwd* or to do

```
ex `grep -l TRACE *.c`
```

to run the editor *ex* supplying as arguments those files whose names end in '.c' which have the string 'TRACE' in them.*

4.4. Other details not covered here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in its manual section.

The shell has a number of command line option flags mostly of use in writing UNIX programs, and debugging shell scripts. See the shells manual section for a list of these options.

*Command expansion also occurs in input redirected with '<<' and within '"' quotations. Refer to the shell manual section for full details.

Appendix — Special characters

The following table lists the special characters of *csh* and the UNIX system, giving for each the section(s) in which it is discussed. A number of these characters also have special meaning in expressions. See the *csh* manual section for a complete list.

Syntactic metacharacters

;	2.4	separates commands to be executed sequentially
|	1.5	separates commands in a pipeline
( )	2.2,3.6	brackets expressions and variable values
&	2.5	follows commands to be executed without waiting for completion

Filename metacharacters

/	1.6	separates components of a file's pathname
?	1.6	expansion character matching any single character
*	1.6	expansion character matching any sequence of characters
[ ]	1.6	expansion sequence matching any single character from a set
~	1.6	used at the beginning of a filename to indicate home directories
{ }	4.2	used to specify groups of arguments with common parts

Quotation metacharacters

\	1.7	prevents meta-meaning of following single character
'	1.7	prevents meta-meaning of a group of characters
"	4.3	like ', but allows variable and command expansion

Input/output metacharacters

<	1.5	indicates redirected input
>	1.3	indicates redirected output

Expansion/substitution metacharacters

$	3.4	indicates variable substitution
!	2.3	indicates history substitution
:	3.6	precedes substitution modifiers
↑	2.3	used in special forms of history substitution
`	4.3	indicates command substitution

Other metacharacters

#	1.3,3.6	begins scratch file names; indicates shell comments
−	1.2	prefixes option (flag) arguments to commands
%	2.6	prefixes job name specifications

Glossary

This glossary lists the most important terms introduced in the introduction to the shell and gives references to sections of the shell document for further information about them References of the form 'pr (1)' indicate that the command *pr* is in the UNIX programmer's manual in section 1. You can get an online copy of its manual page by doing

```
man 1 pr
```

References of the form (2.5) indicate that more information can be found in section 2.5 of this manual.

.	Your current directory has the name '.' as well as the name printed by the command *pwd;* see also *dirs.* The current directory '.' is usually the first *component* of the search path contained in the variable *path,* thus commands which are in '.' are found first (2.2). The character '.' is also used in separating *components* of filenames (1.6). The character '.' at the beginning of a *component* of a *pathname* is treated specially and not matched by the *filename expansion* metacharacters '?', '*', and '[' ']' pairs (1.6).

..	Each directory has a file '..' in it which is a reference to its parent directory. After changing into the directory with *chdir,* i.e.
	`chdir paper`
	you can return to the parent directory by doing
	`chdir ..`
	The current directory is printed by *pwd* (2.7).

a.out	Compilers which create executable images create them, by default, in the file *a.out.* for historical reasons (2.3).
absolute pathname	A *pathname* which begins with a '/' is *absolute* since it specifies the *path* of directories from the beginning of the entire directory system — called the *root* directory. *Pathnames* which are not *absolute* are called *relative* (see definition of *relative pathname*) (1.6).
alias	An *alias* specifies a shorter or different name for a UNIX command, or a transformation on a command to be performed in the shell. The shell has a command *alias* which establishes *aliases* and can print their current values. The command *unalias* is used to remove *aliases* (2.4).
argument	Commands in UNIX receive a list of *argument* words. Thus the command `echo a b c` consists of the *command name* 'echo' and three *argument* words 'a', 'b' and 'c'. The set of *arguments* after the *command name* is said to be the *argument list* of the command (1.1).
argv	The list of arguments to a command written in the shell language (a shell script or shell procedure) is stored in a variable called *argv* within the shell. This name is taken from the conventional name in the C programming language (3.4).
background	Commands started without waiting for them to complete are called *background* commands (2.6).
base	A filename is sometimes thought of as consisting of a *base* part, before any '.' character, and an *extension* — the part after the '.'. See *filename* and *extension* (1.6)

bg	The *bg* command causes a *suspended* job to continue execution in the *background* (2.6).
bin	A directory containing binaries of programs and shell scripts to be executed is typically called a *bin* directory. The standard system *bin* directories are '/bin' containing the most heavily used commands and '/usr/bin' which contains most other user programs. Programs developed at UC Berkeley live in '/usr/ucb', while locally written programs live in '/usr/local'. Games are kept in the directory '/usr/games'. You can place binaries in any directory. If you wish to execute them often, the name of the directories should be a *component* of the variable *path*.
break	*Break* is a builtin command used to exit from loops within the control structure of the shell (3.7).
breaksw	The *breaksw* builtin command is used to exit from a *switch* control structure, like a *break* exits from loops (3.7).
builtin	A command executed directly by the shell is called a *builtin* command. Most commands in UNIX are not built into the shell, but rather exist as files in *bin* directories. These commands are accessible because the directories in which they reside are named in the *path* variable.
case	A *case* command is used as a label in a *switch* statement in the shell's control structure, similar to that of the language C. Details are given in the shell documentation 'csh(1)' (3.7).
cat	The *cat* program catenates a list of specified files on the *standard output*. It is usually used to look at the contents of a single file on the terminal, to 'cat a file' (1.8, 2.3).
cd	The *cd* command is used to change the *working directory*. With no arguments, *cd* changes your *working directory* to be your *home* directory (2.4, 2.7).
chdir	The *chdir* command is a synonym for *cd*. *Cd* is usually used because it is easier to type.

chsh	The *chsh* command is used to change the shell which you use on UNIX. By default, you use an different version of the shell which resides in '/bin/sh'. You can change your shell to '/bin/csh' by doing
	`chsh your-login-name /bin/csh`
	Thus I would do
	`chsh bill /bin/csh`
	It is only necessary to do this once. The next time you log in to UNIX after doing this command, you will be using *csh* rather than the shell in '/bin/sh' (1.9).

cmp	*Cmp* is a program which compares files. It is usually used on binary files, or to see if two files are identical (3.6). For comparing text files the program *diff*, described in 'diff (1)' is used.
command	A function performed by the system, either by the shell (a builtin *command*) or by a program residing in a file in a directory within the UNIX system, is called a *command* (1.1).
command name	When a command is issued, it consists of a *command name*, which is the first word of the command, followed by arguments. The convention on UNIX is that the first word of a command names the function to be performed (1.1).

command substitution	The replacement of a command enclosed in '`' characters by the text output by that command is called *command substitution* (4.3).
component	A part of a *pathname* between '/' characters is called a *component* of that *pathname*. A variable which has multiple strings as value is said to have several *components*; each string is a *component* of the variable.
continue	A builtin command which causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C (3.6).
control-	Certain special characters, called *control* characters, are produced by holding down the CONTROL key on your terminal and simultaneously pressing another character, much like the SHIFT key is used to produce upper case characters. Thus *control-c* is produced by holding down the CONTROL key while pressing the 'c' key. Usually UNIX prints an up-arrow (↑) followed by the corresponding letter when you type a *control* character (e.g. '↑C' for *control-c* (1.8).
core dump	When a program terminates abnormally, the system places an image of its current state in a file named 'core'. This *core dump* can be examined with the system debugger 'adb(1)' or 'sdb(1)' in order to determine what went wrong with the program (1.8). If the shell produces a message of the form Illegal instruction (core dumped) (where 'Illegal instruction' is only one of several possible messages), you should report this to the author of the program or a system administrator, saving the 'core' file.
cp	The *cp* (copy) program is used to copy the contents of one file into another file. It is one of the most commonly used UNIX commands (1.6).
csh	The name of the shell program that this document describes.
.cshrc	The file *.cshrc* in your *home* directory is read by each shell as it begins execution. It is usually used to change the setting of the variable *path* and to set *alias* parameters which are to take effect globally (2.1).
cwd	The *cwd* variable in the shell holds the *absolute pathname* of the current *working directory*. It is changed by the shell whenever your current *working directory* changes and should not be changed otherwise (2.2).
date	The *date* command prints the current date and time (1.3).
debugging	*Debugging* is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell *debugging* (4.4).
default:	The label *default:* is used within shell *switch* statements, as it is in the C language to label the code to be executed if none of the *case* labels matches the value switched on (3.7).
DELETE	The DELETE or RUBOUT key on the terminal normally causes an interrupt to be sent to the current job. Many users change the interrupt character to be ↑C.
detached	A command that continues running in the *background* after you logout is said to be *detached*.
diagnostic	An error message produced by a program is often referred to as a *diagnostic*. Most error messages are not written to the *standard output*, since that is often directed away from the terminal (1.3, 1.5). Error messsages are instead written to the *diagnostic output* which may be directed away from the terminal, but usually is not. Thus *diagnostics* will usually appear on the terminal (2.5).

directory	A structure which contains files. At any time you are in one particular *directory* whose names can be printed by the command *pwd*. The *chdir* command will change you to another *directory*, and make the files in that *directory* visible. The *directory* in which you are when you first login is your *home* directory (1.1, 2.7).
directory stack	The shell saves the names of previous *working directories* in the *directory stack* when you change your current *working directory* via the *pushd* command. The *directory stack* can be printed by using the *dirs* command, which includes your current *working directory* as the first directory name on the left (2.7).
dirs	The *dirs* command prints the shell's *directory stack* (2.7).
du	The *du* command is a program (described in 'du(1)') which prints the number of disk blocks is all directories below and including your current *working directory* (2.6).
echo	The *echo* command prints its arguments (1.6, 3.6).
else	The *else* command is part of the 'if-then-else-endif' control command construct (3.6).
endif	If an *if* statement is ended with the word *then*, all lines following the *if* up to a line starting with the word *endif* or *else* are executed if the condition between parentheses after the *if* is true (3.6).
EOF	An *end-of-file* is generated by the terminal by a control-d, and whenever a command reads to the end of a file which it has been given as input. Commands receiving input from a *pipe* receive an *end-of-file* when the command sending them input completes. Most commands terminate when they receive an *end-of-file*. The shell has an option to ignore *end-of-file* from a terminal input which may help you keep from logging out accidentally by typing too many control-d's (1.1, 1.8, 3.8).

escape	A character '\\' used to prevent the special meaning of a metacharacter is said to *escape* the character from its special meaning. Thus
	
	`echo \*`
	
	will echo the character '*' while just
	
	`echo *`
	
	will echo the names of the file in the current directory. In this example, \\ *escapes* '*' (1.7). There is also a non-printing character called *escape*, usually labelled ESC or ALTMODE on terminal keyboards. Some older UNIX systems use this character to indicate that output is to be *suspended*. Most systems use control-s to stop the output and control-q to start it.

/etc/passwd	This file contains information about the accounts currently on the system. It consists of a line for each account with fields separated by ':' characters (1.8). You can look at this file by saying `cat /etc/passwd` The commands *finger* and *grep* are often used to search for information in this file. See 'finger(1)', 'passwd(5)', and 'grep(1)' for more details.
exit	The *exit* command is used to force termination of a shell script, and is built into the shell (3.9).
exit status	A command which discovers a problem may reflect this back to the command (such as a shell) which invoked (executed) it. It does this by returning a non-zero number as its *exit status*, a status of zero being considered 'normal termination'. The *exit* command can be used to force a shell command script

	to give a non-zero *exit status* (3.6).
expansion	The replacement of strings in the shell input which contain metacharacters by other strings is referred to as the process of *expansion.* Thus the replacement of the word '*' by a sorted list of files in the current directory is a 'filename expansion'. Similarly the replacement of the characters '!!' by the text of the last command is a 'history expansion'. *Expansions* are also referred to as *substitutions* (1.6, 3.4, 4.2).
expressions	*Expressions* are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts. The operators available in shell *expressions* are those of the language C (3.5).
extension	Filenames often consist of a *base* name and an *extension* separated by the character '.'. By convention, groups of related files often share the same *root* name. Thus if 'prog.c' were a C program, then the object file for this program would be stored in 'prog.o'. Similarly a paper written with the '−me' nroff macro package might be stored in 'paper.me' while a formatted version of this paper might be kept in 'paper.out' and a list of spelling errors in 'paper.errs' (1.6).
fg	The *job control* command *fg* is used to run a *background* or *suspended* job in the *foreground* (1.8, 2.6).
filename	Each file in UNIX has a name consisting of up to 14 characters and not including the character '/' which is used in *pathname* building. Most *filenames* do not begin with the character '.', and contain only letters and digits with perhaps a '.' separating the *base* portion of the *filename* from an *extension* (1.6).
filename expansion	*Filename expansion* uses the metacharacters '*', '?' and '[' and ']' to provide a convenient mechanism for naming files. Using *filename expansion* it is easy to name all the files in the current directory, or all files which have a common *root* name. Other *filename expansion* mechanisms use the metacharacter '~' and allow files in other users' directories to be named easily (1.6, 4.2).
flag	Many UNIX commands accept arguments which are not the names of files or other users but are used to modify the action of the commands. These are referred to as *flag* options, and by convention consist of one or more letters preceded by the character '−' (1.2). Thus the *ls* (list files) command has an option '−s' to list the sizes of files. This is specified `ls −s`
foreach	The *foreach* command is used in shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list (3.6, 4.1).
foreground	When commands are executing in the normal way such that the shell is waiting for them to finish before prompting for another command they are said to be *foreground jobs* or *running in the foreground.* This is as opposed to *background. Foreground* jobs can be stopped by signals from the terminal caused by typing different control characters at the keyboard (1.8, 2.6).
goto	The shell has a command *goto* used in shell scripts to transfer control to a given label (3.7).
grep	The *grep* command searches through a list of argument files for a specified string. Thus `grep bill /etc/passwd` will print each line in the file */etc/passwd* which contains the string 'bill'.

	Actually, *grep* scans for *regular expressions* in the sense of the editors 'ed(1)' and 'ex(1)'. *Grep* stands for 'globally find *regular expression* and print' (2.4).
head	The *head* command prints the first few lines of one or more files. If you have a bunch of files containing text which you are wondering about it is sometimes useful to run *head* with these files as arguments. This will usually show enough of what is in these files to let you decide which you are interested in (1.5). *Head* is also used to describe the part of a *pathname* before and including the last '/' character. The *tail* of a *pathname* is the part after the last '/'. The ':h' and ':t' modifiers allow the *head* or *tail* of a *pathname* stored in a shell variable to be used (3.6).
history	The *history* mechanism of the shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a *history list* where these commands are kept, and a *history* variable which controls how large this list is (2.3).
home directory	Each user has a *home directory*, which is given in your entry in the password file, */etc/passwd*. This is the directory which you are placed in when you first login. The *cd* or *chdir* command with no arguments takes you back to this directory, whose name is recorded in the shell variable *home*. You can also access the *home directories* of other users in forming filenames using a *filename expansion* notation and the character '~' (1.6).
if	A conditional command within the shell, the *if* command is used in shell command scripts to make decisions about what course of action to take next (3.6).
ignoreeof	Normally, your shell will exit, printing 'logout' if you type a control-d at a prompt of '% '. This is the way you usually log off the system. You can *set* the *ignoreeof* variable if you wish in your *.login* file and then use the command *logout* to logout. This is useful if you sometimes accidentally type too many control-d characters, logging yourself off (2.2).
input	Many commands on UNIX take information from the terminal or from files which they then act on. This information is called *input*. Commands normally read for *input* from their *standard input* which is, by default, the terminal. This *standard input* can be redirected from a file using a shell metanotation with the character '<'. Many commands will also read from a file specified as argument. Commands placed in *pipelines* will read from the output of the previous command in the *pipeline*. The leftmost command in a *pipeline* reads from the terminal if you neither redirect its *input* nor give it a filename to use as *standard input*. Special mechanisms exist for supplying input to commands in shell scripts (1.5, 3.8).
interrupt	An *interrupt* is a signal to a program that is generated by hitting the RUBOUT or DELETE key (although users can and often do change the interrupt character, usually to ↑C). It causes most programs to stop execution. Certain programs, such as the shell and the editors, handle an *interrupt* in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to *interrupts*. The shell often wakes up when you hit *interrupt* because many commands die when they receive an *interrupt* (1.8, 3.9).
job	One or more commands typed on the same input line separated by '|' or ';' characters are run together and are called a *job*. Simple commands run by themselves without any '|' or ';' characters are the simplest *jobs*. *Jobs* are classified as *foreground*, *background*, or *suspended* (2.6).

job control	The builtin functions that control the execution of jobs are called *job control* commands. These are *bg, fg, stop, kill* (2.6).
job number	When each job is started it is assigned a small number called a *job number* which is printed next to the job in the output of the *jobs* command. This number, preceded by a '%' character, can be used as an argument to *job control* commands to indicate a specific job (2.6).
jobs	The *jobs* command prints a table showing jobs that are either running in the *background* or are *suspended* (2.6).
kill	A command which sends a signal to a job causing it to terminate (2.6).
.login	The file *.login* in your *home* directory is read by the shell each time you login to UNIX and the commands there are executed. There are a number of commands which are usefully placed here, especially *set* commands to the shell itself (2.1).
login shell	The shell that is started on your terminal when you login is called your *login shell*. It is different from other shells which you may run (e.g. on shell scripts) in that it reads the *.login* file before reading commands from the terminal and it reads the *.logout* file after you logout (2.1).
logout	The *logout* command causes a login shell to exit. Normally, a login shell will exit when you hit control-d generating an *end-of-file*, but if you have set *ignoreeof* in you *.login* file then this will not work and you must use *logout* to log off the UNIX system (2.8).
.logout	When you log off of UNIX the shell will execute commands from the file *.logout* in your *home* directory after it prints 'logout'.
lpr	The command *lpr* is the line printer daemon. The standard input of *lpr* spooled and printed on the UNIX line printer. You can also give *lpr* a list of filenames as arguments to be printed. It is most common to use *lpr* as the last component of a *pipeline* (2.3).
ls	The *ls* (list files) command is one of the most commonly used UNIX commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful *flag* arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories (1.2).
mail	The *mail* program is used to send and receive messages from other UNIX users (1.1, 2.1).
make	The *make* command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways *make* is easier to use, and more helpful than shell command scripts (3.2).
makefile	The file containing commands for *make* is called *makefile* (3.2).
manual	The *manual* often referred to is the 'UNIX programmer's manual'. It contains a number of sections and a description of each UNIX program. An online version of the *manual* is accessible through the *man* command. Its documentation can be obtained online via `man man`
metacharacter	Many characters which are neither letters nor digits have special meaning either to the shell or to UNIX. These characters are called *metacharacters*. If it is necessary to place these characters in arguments to commands without them having their special meaning then they must be *quoted*. An example of a *metacharacter* is the character '>' which is used to indicate placement of output

	into a file. For the purposes of the *history* mechanism, most unquoted *meta-characters* form separate words (1.4). The appendix to this user's manual lists the *metacharacters* in groups by their function.
mkdir	The *mkdir* command is used to create a new directory.
modifier	Substitutions with the *history* mechanism, keyed by the character '!' or of variables using the metacharacter '$', are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the *modifier* itself. The *command substitution* mechanism can also be used to perform modification in a similar way, but this notation is less clear (3.6).
more	The program *more* writes a file on your terminal allowing you to control how much text is displayed at a time. *More* can move through the file screenful by screenful, line by line, search forward for a string, or start again at the beginning of the file. It is generally the easiest way of viewing a file (1.8).
noclobber	The shell has a variable *noclobber* which may be set in the file *.login* to prevent accidental destruction of files by the '>' output redirection metasyntax of the shell (2.2, 2.5).
noglob	The shell variable *noglob* is set to suppress the *filename expansion* of arguments containing the metacharacters '~', '*', '?', '[' and ']' (3.6).
notify	The *notify* command tells the shell to report on the termination of a specific *background job* at the exact time it occurs as opposed to waiting until just before the next prompt to report the termination. The *notify* variable, if set, causes the shell to always report the termination of *background* jobs exactly when they occur (2.6).
onintr	The *onintr* command is built into the shell and is used to control the action of a shell command script when an *interrupt* signal is received (3.9).
output	Many commands in UNIX result in some lines of text which are called their *output*. This *output* is usually placed on what is known as the *standard output* which is normally connected to the user's terminal. The shell has a syntax using the metacharacter '>' for redirecting the *standard output* of a command to a file (1.3). Using the *pipe* mechanism and the metacharacter '|' it is also possible for the *standard output* of one command to become the *standard input* of another command (1.5). Certain commands such as the line printer daemon *p* do not place their results on the *standard output* but rather in more useful places such as on the line printer (2.3). Similarly the *write* command places its output on another user's terminal rather than its *standard output* (2.3). Commands also have a *diagnostic output* where they write their error messages. Normally these go to the terminal even if the *standard output* has been sent to a file or another command, but it is possible to direct error diagnostics along with *standard output* using a special metanotation (2.5).
pushd	The *pushd* command, which means 'push directory', changes the shell's *working directory* and also remembers the current *working directory* before the change is made, allowing you to return to the same directory via the *popd* command later without retyping its name (2.7).
path	The shell has a variable *path* which gives the names of the directories in which it searches for the commands which it is given. It always checks first to see if the command it is given is built into the shell. If it is, then it need not search for the command as it can do it internally. If the command is not builtin, then the shell searches for a file with the name given in each of the directories in the *path* variable, left to right. Since the normal definition of the *path* variable is

	`path    (. /usr/ucb /bin /usr/bin)`
	 the shell normally looks in the current directory, and then in the standard system directories '/usr/ucb', '/bin' and '/usr/bin' for the named command (2.2). If the command cannot be found the shell will print an error diagnostic. Scripts of shell commands will be executed using another shell to interpret them if they have 'execute' permission set. This is normally true because a command of the form
	 `chmod 755 script`
	 was executed to turn this execute permission on (3.3). If you add new commands to a directory in the *path*, you should issue the command *rehash* (2.2).

pathname	A list of names, separated by '/' characters, forms a *pathname*. Each *component*, between successive '/' characters, names a directory in which the next *component* file resides. *Pathnames* which begin with the character '/' are interpreted relative to the *root* directory in the filesystem. Other *pathnames* are interpreted relative to the current directory as reported by *pwd*. The last component of a *pathname* may name a directory, but usually names a file.
pipeline	A group of commands which are connected together, the *standard output* of each connected to the *standard input* of the next, is called a *pipeline*. The *pipe* mechanism used to connect these commands is indicated by the shell metacharacter '|' (1.5, 2.3).
popd	The *popd* command changes the shell's *working directory* to the directory you most recently left using the *pushd* command. It returns to the directory without having to type its name, forgetting the name of the current *working directory* before doing so (2.7).
port	The part of a computer system to which each terminal is connected is called a *port*. Usually the system has a fixed number of *ports*, some of which are connected to telephone lines for dial-up access, and some of which are permanently wired directly to specific terminals.
pr	The *pr* command is used to prepare listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified (2.3).
printenv	The *printenv* command is used to print the current setting of variables in the environment (2.8).
process	An instance of a running program is called a *process* (2.6). UNIX assigns each *process* a unique number when it is started — called the *process number*. *Process numbers* can be used to stop individual *processes* using the *kill* or *stop* commands when the *processes* are part of a detached *background* job.
program	Usually synonymous with *command*; a binary file or shell command script which performs a useful function is often called a *program*.
programmer's manual	Also referred to as the *manual*. See the glossary entry for 'manual'.
prompt	Many programs will print a *prompt* on the terminal when they expect input. Thus the editor 'ex(1)' will print a ':' when it expects input. The shell *prompts* for input with '% ' and occasionally with '? ' when reading commands from the terminal (1.1). The shell has a variable *prompt* which may be set to a different value to change the shell's main *prompt*. This is mostly used when debugging the shell (2.8).

ps	The *ps* command is used to show the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, an indication of the state of the process (whether it is running, stopped, awaiting some event (sleeping), and whether it is swapped out), and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked (2.6). Shells, such as the *csh* you use to run the *ps* command, are not normally shown in the output.
pwd	The *pwd* command prints the full *pathname* of the current *working directory*. The *dirs* builtin command is usually a better and faster choice.
quit	The *quit* signal, generated by a control-\, is used to terminate programs which are behaving unreasonably. It normally produces a core image file (1.8).
quotation	The process by which metacharacters are prevented their special meaning, usually by using the character '' in pairs, or by using the character '\', is referred to as *quotation* (1.7).
redirection	The routing of input or output from or to a file is known as *redirection* of input or output (1.3).
rehash	The *rehash* command tells the shell to rebuild its internal table of which commands are found in which directories in your *path*. This is necessary when a new program is installed in one of these directories (2.8).
relative pathname	A *pathname* which does not begin with a '/' is called a *relative pathname* since it is interpreted *relative* to the current *working directory*. The first *component* of such a *pathname* refers to some file or directory in the *working directory*, and subsequent *components* between '/' characters refer to directories below the *working directory*. *Pathnames* that are not *relative* are called *absolute pathnames* (1.6).
repeat	The *repeat* command iterates another command a specified number of times.
root	The directory that is at the top of the entire directory structure is called the *root* directory since it is the 'root' of the entire tree structure of directories. The name used in *pathnames* to indicate the *root* is '/'. *Pathnames* starting with '/' are said to be *absolute* since they start at the *root* directory. *Root* is also used as the part of a *pathname* that is left after removing the *extension*. See *filename* for a further explanation (1.6).
RUBOUT	The RUBOUT or DELETE key sends an interrupt to the current job. Most interactive commands return to their command level upon receipt of an interrupt, while non-interactive commands usually terminate, returning control to the shell. Users often change interrupt to be generated by ↑C rather than DELETE by using the *stty* command.
scratch file	Files whose names begin with a '#' are referred to as *scratch files*, since they are automatically removed by the system after a couple of days of non-use, or more frequently if disk space becomes tight (1.3).
script	Sequences of shell commands placed in a file are called shell command *scripts*. It is often possible to perform simple tasks using these *scripts* without writing a program in a language such as C, by using the shell to selectively run other programs (3.3, 3.10).
set	The builtin *set* command is used to assign new values to shell variables and to show the values of the current variables. Many shell variables have special meaning to the shell itself. Thus by using the *set* command the behavior of the shell can be affected (2.1).

setenv	Variables in the environment 'environ(5)' can be changed by using the *setenv* builtin command (2.8). The *printenv* command can be used to print the value of the variables in the environment.
shell	A *shell* is a command language interpreter. It is possible to write and run your own *shell*, as *shells* are no different than any other programs as far as the system is concerned. This manual deals with the details of one particular *shell*, called *csh*.
shell script	See *script* (3.3, 3.10).
signal	A *signal* in UNIX is a short message that is sent to a running program which causes something to happen to that process. *Signals* are sent either by typing special *control* characters on the keyboard or by using the *kill* or *stop* commands (1.8, 2.6).
sort	The *sort* program sorts a sequence of lines in ways that can be controlled by argument *flags* (1.5).
source	The *source* command causes the shell to read commands from a specified file. It is most useful for reading files such as *.cshrc* after changing them (2.8).
special character	See *metacharacters* and the appendix to this manual.
standard	We refer often to the *standard input* and *standard output* of commands. See *input* and *output* (1.3, 3.8).
status	A command normally returns a *status* when it finishes. By convention a *status* of zero indicates that the command succeeded. Commands may return non-zero *status* to indicate that some abnormal event has occurred. The shell variable *status* is set to the *status* returned by the last command. It is most useful in shell commmand scripts (3.6).
stop	The *stop* command causes a *background* job to become *suspended* (2.6).
string	A sequential group of characters taken together is called a *string*. *Strings* can contain any printable characters (2.2).
stty	The *stty* program changes certain parameters inside UNIX which determine how your terminal is handled. See 'stty(1)' for a complete description (2.6).
substitution	The shell implements a number of *substitutions* where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history *substitution* keyed by the metacharacter '!' and variable *substitution* indicated by '$'. We also refer to *substitutions* as *expansions* (3.4).
suspended	A job becomes *suspended* after a STOP signal is sent to it, either by typing a *control*-z at the terminal (for *foreground* jobs) or by using the *stop* command (for *background* jobs). When *suspended*, a job temporarily stops running until it is restarted by either the *fg* or *bg* command (2.6).
switch	The *switch* command of the shell allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the *switch* statement in the language C (3.7).
termination	When a command which is being executed finishes we say it undergoes *termination* or *terminates*. Commands normally terminate when they read an *end-of-file* from their *standard input*. It is also possible to terminate commands by sending them an *interrupt* or *quit* signal (1.8). The *kill* program terminates specified jobs (2.6).
then	The *then* command is part of the shell's 'if-then-else-endif' control construct used in command scripts (3.6).

time	The *time* command can be used to measure the amount of CPU and real time consumed by a specified command as well as the amount of disk i/o, memory utilized, and number of page faults and swaps taken by the command (2.1, 2.8).
tset	The *tset* program is used to set standard erase and kill characters and to tell the system what kind of terminal you are using. It is often invoked in a *.login* file (2.1).
tty	The word *tty* is a historical abbreviation for 'teletype' which is frequently used in UNIX to indicate the *port* to which a given terminal is connected. The *tty* command will print the name of the *tty* or *port* to which your terminal is presently connected.
unalias	The *unalias* command removes aliases (2.8).
UNIX	UNIX is an operating system on which *csh* runs. UNIX provides facilities which allow *csh* to invoke other programs such as editors and text formatters which you may wish to use.
unset	The *unset* command removes the definitions of shell variables (2.2, 2.8).
variable expansion	See *variables* and *expansion* (2.2, 3.4).
variables	*Variables* in *csh* hold one or more strings as value. The most common use of *variables* is in controlling the behavior of the shell. See *path*, *noclobber*, and *ignoreeof* for examples. *Variables* such as *argv* are also used in writing shell programs (shell command scripts) (2.2).
verbose	The *verbose* shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging shell scripts. The *verbose* variable is set by the shell's *−v* command line option (3.10).
wc	The *wc* program calculates the number of characters, words, and lines in the files whose names are given as arguments (2.6).
while	The *while* builtin control construct is used in shell command scripts (3.7).
word	A sequence of characters which forms an argument to a command is called a *word*. Many characters which are neither letters, digits, '−', '.' nor '/' form *words* all by themselves even if they are not surrounded by blanks. Any sequence of characters may be made into a *word* by surrounding it with ''' characters except for the characters ''' and '!' which require special treatment (1.1). This process of placing special characters in *words* without their special meaning is called *quoting*.
working directory	At any given time you are in one particular directory, called your *working directory*. This directory's name is printed by the *pwd* command and the files listed by *ls* are the ones in this directory. You can change *working directories* using *chdir*.
write	The *write* command is used to communicate with other users who are logged in to UNIX.

MAIL REFERENCE MANUAL

Kurt Shoens

Revised by

Craig Leres

Version 2.18

July 27, 1983

1. Introduction

Mail provides a simple and friendly environment for sending and receiving mail. It divides incoming mail into its constituent messages and allows the user to deal with them in any order. In addition, it provides a set of *ed*-like commands for manipulating messages and sending mail. *Mail* offers the user simple editing capabilities to ease the composition of outgoing messages, as well as providing the ability to define and send to names which address groups of users. Finally, *Mail* is able to send and receive messages across such networks as the ARPANET, UUCP, and Berkeley network.

This document describes how to use the *Mail* program to send and receive messages. The reader is not assumed to be familiar with other message handling systems, but should be familiar with the UNIX[1] shell, the text editor, and some of the common UNIX commands. "The UNIX Programmer's Manual," "An Introduction to Csh," and "Text Editing with Ex and Vi" can be consulted for more information on these topics.

Here is how messages are handled: the mail system accepts incoming *messages* for you from other people and collects them in a file, called your *system mailbox.* When you login, the system notifies you if there are any messages waiting in your system mailbox. If you are a *csh* user, you will be notified when new mail arrives if you inform the shell of the location of your mailbox. On version 7 systems, your system mailbox is located in the directory /usr/spool/mail in a file with your login name. If your login name is "sam," then you can make *csh* notify you of new mail by including the following line in your .cshrc file:

```
set mail=/usr/spool/mail/sam
```

When you read your mail using *Mail,* it reads your system mailbox and separates that file into the individual messages that have been sent to you. You can then read, reply to, delete, or save these messages. Each message is marked with its author and the date they sent it.

[1] UNIX is a trademark of Bell Laboratories.

2. Common usage

The *Mail* command has two distinct usages, according to whether one wants to send or receive mail. Sending mail is simple: to send a message to a user whose login name is, say, "root," use the shell command:

```
% Mail root
```

then type your message. When you reach the end of the message, type an EOT (control−d) at the beginning of a line, which will cause *Mail* to echo "EOT" and return you to the Shell. When the user you sent mail to next logs in, he will receive the message:

```
You have mail.
```

to alert him to the existence of your message.

If, while you are composing the message you decide that you do not wish to send it after all, you can abort the letter with a RUBOUT. Typing a single RUBOUT causes *Mail* to print

```
(Interrupt -- one more to kill letter)
```

Typing a second RUBOUT causes *Mail* to save your partial letter on the file "dead.letter" in your home directory and abort the letter. Once you have sent mail to someone, there is no way to undo the act, so be careful.

The message your recipient reads will consist of the message you typed, preceded by a line telling who sent the message (your login name) and the date and time it was sent.

If you want to send the same message to several other people, you can list their login names on the command line. Thus,

```
% Mail sam bob john
Tuition fees are due next Friday.  Don't forget!!
<Control−d>
EOT
%
```

will send the reminder to sam, bob, and john.

If, when you log in, you see the message,

```
You have mail.
```

you can read the mail by typing simply:

```
% Mail
```

Mail will respond by typing its version number and date and then listing the messages you have waiting. Then it will type a prompt and await your command. The messages are assigned numbers starting with 1 — you refer to the messages with these numbers. *Mail* keeps tack of which messages are *new* (have been sent since you last read your mail) and *read* (have been read by you). New messages have an N next to them in the header listing and old, but unread messages have a U next to them. *Mail* keeps track of new/old and read/unread messages by putting a header field called "Status" into your messages.

To look at a specific message, use the **type** command, which may be abbreviated to simply **t**. For example, if you had the following messages:

```
N 1 root     Wed Sep 21 09:21  "Tuition fees"
N 2 sam      Tue Sep 20 22:55
```

you could examine the first message by giving the command:

```
type 1
```

which might cause *Mail* to respond with, for example:

```
Message  1:
From root  Wed Sep 21 09:21:45 1978
Subject: Tuition fees
```

```
Status: R

Tuition fees are due next Wednesday.  Don't forget!!
```

Many *Mail* commands that operate on messages take a message number as an argument like the **type** command. For these commands, there is a notion of a current message. When you enter the *Mail* program, the current message is initially the first one. Thus, you can often omit the message number and use, for example,

```
t
```

to type the current message. As a further shorthand, you can type a message by simply giving its message number. Hence,

```
1
```

would type the first message.

Frequently, it is useful to read the messages in your mailbox in order, one after another. You can read the next message in *Mail* by simply typing a newline. As a special case, you can type a newline as your first command to *Mail* to type the first message.

If, after typing a message, you wish to immediately send a reply, you can do so with the **reply** command. **Reply**, like **type**, takes a message number as an argument. *Mail* then begins a message addressed to the user who sent you the message. You may then type in your letter in reply, followed by a <control-d> at the beginning of a line, as before. *Mail* will type EOT, then type the ampersand prompt to indicate its readiness to accept another command. In our example, if, after typing the first message, you wished to reply to it, you might give the command:

```
reply
```

Mail responds by typing:

```
To: root
Subject: Re: Tuition fees
```

and waiting for you to enter your letter. You are now in the message collection mode described at the beginning of this section and *Mail* will gather up your message up to a control−d. Note that it copies the subject header from the original message. This is useful in that correspondence about a particular matter will tend to retain the same subject heading, making it easy to recognize. If there are other header fields in the message, the information found will also be used. For example, if the letter had a "To:" header listing several recipients, *Mail* would arrange to send your replay to the same people as well. Similarly, if the original message contained a "Cc:" (carbon copies to) field, *Mail* would send your reply to *those* users, too. *Mail* is careful, though, not too send the message to *you*, even if you appear in the "To:" or "Cc:" field, unless you ask to be included explicitly. See section 4 for more details.

After typing in your letter, the dialog with *Mail* might look like the following:

```
reply
To: root
Subject: Tuition fees

Thanks for the reminder
EOT
&
```

The **reply** command is especially useful for sustaining extended conversations over the message system, with other "listening" users receiving copies of the conversation. The **reply** command can be abbreviated to **r**.

Sometimes you will receive a message that has been sent to several people and wish to reply *only* to the person who sent it. **Reply** with a capital **R** replies to a message, but sends a copy to the sender only.

If you wish, while reading your mail, to send a message to someone, but not as a reply to one of your messages, you can send the message directly with the **mail** command, which takes as arguments the names of the recipients you wish to send to. For example, to send a message to "frank," you would do:

```
mail frank
This is to confirm our meeting next Friday at 4.
EOT
&
```

The **mail** command can be abbreviated to **m**.

Normally, each message you receive is saved in the file *mbox* in your login directory at the time you leave *Mail.* Often, however, you will not want to save a particular message you have received because it is only of passing interest. To avoid saving a message in *mbox* you can delete it using the **delete** command. In our example,

```
delete 1
```

will prevent *Mail* from saving message 1 (from root) in *mbox.* In addition to not saving deleted messages, *Mail* will not let you type them, either. The effect is to make the message disappear altogether, along with its number. The **delete** command can be abbreviated to simply **d**.

Many features of *Mail* can be tailored to your liking with the **set** command. The **set** command has two forms, depending on whether you are setting a *binary* option or a *valued* option. Binary options are either on or off. For example, the "ask" option informs *Mail* that each time you send a message, you want it to prompt you for a subject header, to be included in the message. To set the "ask" option, you would type

```
set ask
```

Another useful *Mail* option is "hold." Unless told otherwise, *Mail* moves the messages from your system mailbox to the file *mbox* in your home directory when you leave *Mail.* If you want *Mail* to keep your letters in the system mailbox instead, you can set the "hold" option.

Valued options are values which *Mail* uses to adapt to your tastes. For example, the "SHELL" option tells *Mail* which shell you like to use, and is specified by

```
set SHELL=/bin/csh
```

for example. Note that no spaces are allowed in "SHELL=/bin/csh." A complete list of the *Mail* options appears in section 5.

Another important valued option is "crt." If you use a fast video terminal, you will find that when you print long messages, they fly by too quickly for you to read them. With the "crt" option, you can make *Mail* print any message larger than a given number of lines by sending it through the paging program *more.* For example, most CRT users should do:

```
set crt=24
```

to paginate messages that will not fit on their screens. *More* prints a screenful of information, then types --MORE--. Type a space to see the next screenful.

Another adaptation to user needs that *Mail* provides is that of *aliases.* An alias is simply a name which stands for one or more real user names. *Mail* sent to an alias is really sent to the list of real users associated with it. For example, an alias can be defined for the members of a project, so that you can send mail to the whole project by sending mail to just a single name. The **alias** command in *Mail* defines an alias. Suppose that the users in a project are named Sam, Sally, Steve, and Susan. To define an alias called "project" for them, you would use the *Mail* command:

```
alias project sam sally steve susan
```

The **alias** command can also be used to provide a convenient name for someone whose user name is inconvenient. For example, if a user named "Bob Anderson" had the login name "anderson,"" you might want to use:

    alias bob anderson

so that you could send mail to the shorter name, "bob."

While the **alias** and **set** commands allow you to customize *Mail*, they have the drawback that they must be retyped each time you enter *Mail*. To make them more convenient to use, *Mail* always looks for two files when it is invoked. It first reads a system wide file "/usr/lib/Mail.rc," then a user specific file, ".mailrc," which is found in the user's home directory. The system wide file is maintained by the system administrator and contains **set** commands that are applicable to all users of the system. The ".mailrc" file is usually used by each user to set options the way he likes and define individual aliases. For example, my .mailrc file looks like this:

    set ask nosave SHELL=/bin/csh

As you can see, it is possible to set many options in the same **set** command. The "nosave" option is described in section 5.

Mail aliasing is implemented at the system-wide level by the mail delivery system *sendmail*. These aliases are stored in the file /usr/lib/aliases and are accessible to all users of the system. The lines in /usr/lib/aliases are of the form:

alias: name_1, name_2, name_3

where *alias* is the mailing list name and the $name_i$ are the members of the list. Long lists can be continued onto the next line by starting the next line with a space or tab. Remember that you must execute the shell command *newaliases* after editing /usr/lib/aliases since the delivery system uses an indexed file created by *newaliases*.

We have seen that *Mail* can be invoked with command line arguments which are people to send the message to, or with no arguments to read mail. Specifying the **−f** flag on the command line causes *Mail* to read messages from a file other than your system mailbox. For example, if you have a collection of messages in the file "letters" you can use *Mail* to read them with:

    % Mail −f letters

You can use all the *Mail* commands described in this document to examine, modify, or delete messages from your "letters" file, which will be rewritten when you leave *Mail* with the **quit** command described below.

Since mail that you read is saved in the file *mbox* in your home directory by default, you can read *mbox* in your home directory by using simply

    % Mail −f

Normally, messages that you examine using the **type** command are saved in the file "mbox" in your home directory if you leave *Mail* with the **quit** command described below. If you wish to retain a message in your system mailbox you can use the **preserve** command to tell *Mail* to leave it there. The **preserve** command accepts a list of message numbers, just like **type** and may be abbreviated to **pre**.

Messages in your system mailbox that you do not examine are normally retained in your system mailbox automatically. If you wish to have such a message saved in *mbox* without reading it, you may use the **mbox** command to have them so saved. For example,

    mbox 2

in our example would cause the second message (from sam) to be saved in *mbox* when the **quit** command is executed. **Mbox** is also the way to direct messages to your *mbox* file if you have set the "hold" option described above. **Mbox** can be abbreviated to **mb**.

When you have perused all the messages of interest, you can leave *Mail* with the **quit** command, which saves the messages you have typed but not deleted in the file *mbox* in your login directory. Deleted messages are discarded irretrievably, and messages left untouched are preserved in your system mailbox so that you will see them the next time you type:

```
% Mail
```

The **quit** command can be abbreviated to simply **q**.

If you wish for some reason to leave *Mail* quickly without altering either your system mailbox or *mbox*, you can type the **x** command (short for **exit**), which will immediately return you to the Shell without changing anything.

If, instead, you want to execute a Shell command without leaving *Mail*, you can type the command preceded by an exclamation point, just as in the text editor. Thus, for instance:

```
!date
```

will print the current date without leaving *Mail*.

Finally, the **help** command is available to print out a brief summary of the *Mail* commands, using only the single character command abbreviations.

3. Maintaining folders

Mail includes a simple facility for maintaining groups of messages together in folders. This section describes this facility.

To use the folder facility, you must tell *Mail* where you wish to keep your folders. Each folder of messages will be a single file. For convenience, all of your folders are kept in a single directory of your choosing. To tell *Mail* where your folder directory is, put a line of the form

```
set folder=letters
```

in your *.mailrc* file. If, as in the example above, your folder directory does not begin with a '/,' *Mail* will assume that your folder directory is to be found starting from your home directory. Thus, if your home directory is **/usr/person** the above example told *Mail* to find your folder directory in **/usr/person/letters**.

Anywhere a file name is expected, you can use a folder name, preceded with '+.' For example, to put a message into a folder with the **save** command, you can use:

```
save +classwork
```

to save the current message in the *classwork* folder. If the *classwork* folder does not yet exist, it will be created. Note that messages which are saved with the **save** command are automatically removed from your system mailbox.

In order to make a copy of a message in a folder without causing that message to be removed from your system mailbox, use the **copy** command, which is identical in all other respects to the **save** command. For example,

```
copy +classwork
```

copies the current message into the *classwork* folder and leaves a copy in your system mailbox.

The **folder** command can be used to direct *Mail* to the contents of a different folder. For example,

```
folder +classwork
```

directs *Mail* to read the contents of the *classwork* folder. All of the commands that you can use on your system mailbox are also applicable to folders, including **type**, **delete**, and **reply**. To inquire which folder you are currently editing, use simply:

```
folder
```

To list your current set of folders, use the **folders** command.

To start *Mail* reading one of your folders, you can use the **−f** option described in section 2. For example:

```
% Mail −f +classwork
```

will cause *Mail* to read your *classwork* folder without looking at your system mailbox.

4. More about sending mail

4.1. Tilde escapes

While typing in a message to be sent to others, it is often useful to be able to invoke the text editor on the partial message, print the message, execute a shell command, or do some other auxiliary function. *Mail* provides these capabilities through *tilde escapes*, which consist of a tilde (~) at the beginning of a line, followed by a single character which indicates the function to be performed. For example, to print the text of the message so far, use:

```
~p
```

which will print a line of dashes, the recipients of your message, and the text of the message so far. Since *Mail* requires two consecutive RUBOUT's to abort a letter, you can use a single RUBOUT to abort the output of ~p or any other ~ escape without killing your letter.

If you are dissatisfied with the message as it stands, you can invoke the text editor on it using the escape

```
~e
```

which causes the message to be copied into a temporary file and an instance of the editor to be spawned. After modifying the message to your satisfaction, write it out and quit the editor. *Mail* will respond by typing

```
(continue)
```

after which you may continue typing text which will be appended to your message, or type <control-d> to end the message. A standard text editor is provided by *Mail*. You can override this default by setting the valued option "EDITOR" to something else. For example, you might prefer:

```
set EDITOR=/usr/ucb/ex
```

Many systems offer a screen editor as an alternative to the standard text editor, such as the *vi* editor from UC Berkeley. To use the screen, or *visual* editor, on your current message, you can use the escape,

```
~v
```

~v works like ~e, except that the screen editor is invoked instead. A default screen editor is defined by *Mail*. If it does not suit you, you can set the valued option "VISUAL" to the path name of a different editor.

It is often useful to be able to include the contents of some file in your message; the escape

```
~r filename
```

is provided for this purpose, and causes the named file to be appended to your current message. *Mail* complains if the file doesn't exist or can't be read. If the read is successful, the number of lines and characters appended to your message is printed, after which you may continue appending text. The filename may contain shell metacharacters like * and ? which are expanded according to the conventions of your shell.

As a special case of ~r, the escape

```
~d
```

reads in the file "dead.letter" in your home directory. This is often useful since *Mail* copies the text of your message there when you abort a message with RUBOUT.

To save the current text of your message on a file you may use the

```
~w filename
```

escape. *Mail* will print out the number of lines and characters written to the file, after which you may continue appending text to your message. Shell metacharacters may be used in the filename, as in ~r and are expanded with the conventions of your shell.

If you are sending mail from within *Mail's* command mode you can read a message sent to you into the message you are constructing with the escape:

```
~m 4
```

which will read message 4 into the current message, shifted right by one tab stop. You can name any non-deleted message, or list of messages. Messages can also be forwarded without shifting by a tab stop with ~f. This is the usual way to forward a message.

If, in the process of composing a message, you decide to add additional people to the list of message recipients, you can do so with the escape

```
~t name1 name2 ...
```

You may name as few or many additional recipients as you wish. Note that the users originally on the recipient list will still receive the message; you cannot remove someone from the recipient list with ~t.

If you wish, you can associate a subject with your message by using the escape

```
~s Arbitrary string of text
```

which replaces any previous subject with "Arbitrary string of text." The subject, if given, is sent near the top of the message prefixed with "Subject:" You can see what the message will look like by using ~p.

For political reasons, one occasionally prefers to list certain people as recipients of carbon copies of a message rather than direct recipients. The escape

```
~c name1 name2 ...
```

adds the named people to the "Cc:" list, similar to ~t. Again, you can execute ~p to see what the message will look like.

The recipients of the message together constitute the "To:" field, the subject the "Subject:" field, and the carbon copies the "Cc:" field. If you wish to edit these in ways impossible with the ~t, ~s, and ~c escapes, you can use the escape

```
~h
```

which prints "To:" followed by the current list of recipients and leaves the cursor (or printhead) at the end of the line. If you type in ordinary characters, they are appended to the end of the current list of recipients. You can also use your erase character to erase back into the list of recipients, or your kill character to erase them altogether. Thus, for example, if your erase and kill characters are the standard # and @ symbols,

```
~h
To: root kurt####bill
```

would change the initial recipients "root kurt" to "root bill." When you type a newline, *Mail* advances to the "Subject:" field, where the same rules apply. Another newline brings you to the "Cc:" field, which may be edited in the same fashion. Another newline leaves you appending text to the end of your message. You can use ~p to print the current text of the header fields and the body of the message.

To effect a temporary escape to the shell, the escape

```
~!command
```

is used, which executes *command* and returns you to mailing mode without altering the text of your message. If you wish, instead, to filter the body of your message through a shell command, then you can use

```
~|command
```

which pipes your message through the command and uses the output as the new text of your message. If the command produces no output, *Mail* assumes that something is amiss and retains the old version of your message. A frequently-used filter is the command *fmt*, designed to format outgoing mail.

To effect a temporary escape to *Mail* command mode instead, you can use the

~:*Mail command*

escape. This is especially useful for retyping the message you are replying to, using, for example:

~:t

It is also useful for setting options and modifying aliases.

If you wish (for some reason) to send a message that contains a line beginning with a tilde, you must double it. Thus, for example,

~~This line begins with a tilde.

sends the line

~This line begins with a tilde.

Finally, the escape

~?

prints out a brief summary of the available tilde escapes.

On some terminals (particularly ones with no lower case) tilde's are difficult to type. *Mail* allows you to change the escape character with the "escape" option. For example, I set

set escape=]

and use a right bracket instead of a tilde. If I ever need to send a line beginning with right bracket, I double it, just as for ~. Changing the escape character removes the special meaning of ~..

4.2. Network access

This section describes how to send mail to people on other machines. Recall that sending to a plain login name sends mail to that person on your machine. If your machine is directly (or sometimes, even, indirectly) connected to the Arpanet, you can send messages to people on the Arpanet using a name of the form

name@host

where *name* is the login name of the person you're trying to reach and *host* is the name of the machine where he logs in on the Arpanet.

If your recipient logs in on a machine connected to yours by UUCP (the Bell Laboratories supplied network that communicates over telephone lines), sending mail to him is a bit more complicated. You must know the list of machines through which your message must travel to arrive at his site. So, if his machine is directly connected to yours, you can send mail to him using the syntax:

host!name

where, again, *host* is the name of his machine and *name* is his login name. If your message must go through an intermediate machine first, you must use the syntax:

intermediate!host!name

and so on. It is actually a feature of UUCP that the map of all the systems in the network is not known anywhere (except where people decide to write it down for convenience). Talk to your system administrator about the machines connected to your site.

If you want to send a message to a recipient on the Berkeley network (Berknet), you use the syntax:

host:name

where *host* is his machine name and *name* is his login name. Unlike UUCP, you need not know the names of the intermediate machines.

When you use the **reply** command to respond to a letter, there is a problem of figuring out the names of the users in the "To:" and "Cc:" lists *relative to the current machine.* If the original letter was sent to you by someone on the local machine, then this problem does not exist, but if the message came from a remote machine, the problem must be dealt with. *Mail* uses a heuristic to build the correct name for each user relative to the local machine. So, when you **reply** to remote mail, the names in the "To:" and "Cc:" lists may change somewhat.

4.3. Special recipients

As described previously, you can send mail to either user names or **alias** names. It is also possible to send messages directly to files or to programs, using special conventions. If a recipient name has a '/' in it or begins with a '+', it is assumed to be the path name of a file into which to send the message. If the file already exists, the message is appended to the end of the file. If you want to name a file in your current directory (ie, one for which a '/' would not usually be needed) you can precede the name with './' So, to send mail to the file "memo" in the current directory, you can give the command:

```
% Mail ./memo
```

If the name begins with a '+,' it is expanded into the full path name of the folder name in your folder directory. This ability to send mail to files can be used for a variety of purposes, such as maintaining a journal and keeping a record of mail sent to a certain group of users. The second example can be done automatically by including the full pathname of the record file in the **alias** command for the group. Using our previous **alias** example, you might give the command:

```
alias project sam sally steve susan /usr/project/mail_record
```

Then, all mail sent to "project" would be saved on the file "/usr/project/mail_record" as well as being sent to the members of the project. This file can be examined using *Mail −f.*

It is sometimes useful to send mail directly to a program, for example one might write a project billboard program and want to access it using *Mail.* To send messages to the billboard program, one can send mail to the special name '|billboard' for example. *Mail* treats recipient names that begin with a '|' as a program to send the mail to. An **alias** can be set up to reference a '|' prefaced name if desired. *Caveats*: the shell treats '|' specially, so it must be quoted on the command line. Also, the '| program' must be presented as a single argument to mail. The safest course is to surround the entire name with double quotes. This also applies to usage in the **alias** command. For example, if we wanted to alias 'rmsgs' to 'rmsgs −s' we would need to say:

```
alias rmsgs "| rmsgs -s"
```

5. Additional features

This section describes some additional commands of use for reading your mail, setting options, and handling lists of messages.

5.1. Message lists

Several *Mail* commands accept a list of messages as an argument. Along with **type** and **delete**, described in section 2, there is the **from** command, which prints the message headers associated with the message list passed to it. The **from** command is particularly useful in conjunction with some of the message list features described below.

A *message list* consists of a list of message numbers, ranges, and names, separated by spaces or tabs. Message numbers may be either decimal numbers, which directly specify messages, or one of the special characters "↑" "." or "$" to specify the first relevant, current, or last relevant message, respectively. *Relevant* here means, for most commands "not deleted" and "deleted" for the **undelete** command.

A range of messages consists of two message numbers (of the form described in the previous paragraph) separated by a dash. Thus, to print the first four messages, use

```
type 1−4
```

and to print all the messages from the current message to the last message, use

```
type .−$
```

A *name* is a user name. The user names given in the message list are collected together and each message selected by other means is checked to make sure it was sent by one of the named users. If the message consists entirely of user names, then every message sent by one those users that is *relevant* (in the sense described earlier) is selected. Thus, to print every message sent to you by "root," do

```
type root
```

As a shorthand notation, you can specify simply "*" to get every *relevant* (same sense) message. Thus,

```
type *
```

prints all undeleted messages,

```
delete *
```

deletes all undeleted messages, and

```
undelete *
```

undeletes all deleted messages.

You can search for the presence of a word in subject lines with /. For example, to print the headers of all messages that contain the word "PASCAL," do:

```
from /pascal
```

Note that subject searching ignores upper/lower case differences.

5.2. List of commands

This section describes all the *Mail* commands available when receiving mail.

! Used to preface a command to be executed by the shell.

− The − command goes to the previous message and prints it. The − command may be given a decimal number *n* as an argument, in which case the *n*th previous message is gone to and printed.

PrintLike **print**, but also print out ignored header fields. See also **print** and **ignore**.

Reply
Note the capital R in the name. Frame a reply to a one or more messages. The reply (or replies if you are using this on multiple messages) will be sent ONLY to the person who sent you the message (respectively, the set of people who sent the messages you are replying to). You can add people using the **~t** and **~c** tilde escapes. The subject in your reply is formed by prefacing the subject in the original message with "Re:" unless it already began thus. If the original message included a "reply-to" header field, the reply will go *only* to the recipient named by "reply-to." You type in your message using the same conventions available to you through the **mail** command. The **Reply** command is especially useful for replying to messages that were sent to enormous distribution groups when you really just want to send a message to the originator. Use it often.

Type Identical to the **Print** command.

alias Define a name to stand for a set of other names. This is used when you want to send messages to a certain group of people and want to avoid retyping their names. For example

alias project john sue willie kathryn

creates an alias *project* which expands to the four people John, Sue, Willie, and Kathryn.

alternates
If you have accounts on several machines, you may find it convenient to use the /usr/lib/aliases on all the machines except one to direct your mail to a single account. The **alternates** command is used to inform *Mail* that each of these other addresses is really *you. Alternates* takes a list of user names and remembers that they are all actually you. When you **reply** to messages that were sent to one of these alternate names, *Mail* will not bother to send a copy of the message to this other address (which would simply be directed back to you by the alias mechanism). If *alternates* is given no argument, it lists the current set of alternate names. **Alternates** is usually used in the .mailrc file.

chdir The **chdir** command allows you to change your current directory. **Chdir** takes a single argument, which is taken to be the pathname of the directory to change to. If no argument is given, **chdir** changes to your home directory.

copy The **copy** command does the same thing that **save** does, except that it does not mark the messages it is used on for deletion when you quit.

delete
Deletes a list of messages. Deleted messages can be reclaimed with the **undelete** command.

dt The **dt** command deletes the current message and prints the next message. It is useful for quickly reading and disposing of mail.

edit To edit individual messages using the text editor, the **edit** command is provided. The **edit** command takes a list of messages as described under the **type** command and processes each by writing it into the file Message*x* where *x* is the message number being edited and executing the text editor on it. When you have edited the message to your satisfaction, write the message out and quit, upon which *Mail* will read the message back and remove the file. **Edit** may be abbreviated to **e**.

else Marks the end of the then-part of an **if** statement and the beginning of the part to take effect if the condition of the **if** statement is false.

endif Marks the end of an **if** statement.

exit Leave *Mail* without updating the system mailbox or the file your were reading. Thus, if you accidentally delete several messages, you can use **exit** to avoid scrambling your mailbox.

file The same as **folder**.

folders
List the names of the folders in your folder directory.

folder
The **folder** command switches to a new mail file or folder. With no arguments, it tells you which file you are currently reading. If you give it an argument, it will write out changes (such as deletions) you have made in the current file and read the new file. Some special conventions are recognized for the name:

Name	Meaning
#	Previous file read
%	Your system mailbox
%name	*Name*'s system mailbox
&	Your ~/mbox file
+folder	A file in your folder directory

from The **from** command takes a list of messages and prints out the header lines for each one; hence

```
from joe
```

is the easy way to display all the message headers from "joe."

headers
When you start up *Mail* to read your mail, it lists the message headers that you have. These headers tell you who each message is from, when they were sent, how many lines and characters each message is, and the "Subject:" header field of each message, if present. In addition, *Mail* tags the message header of each message that has been the object of the **preserve** command with a "P." Messages that have been **saved** or **written** are flagged with a "*." Finally, **deleted** messages are not printed at all. If you wish to reprint the current list of message headers, you can do so with the **headers** command. The **headers** command (and thus the initial header listing) only lists the first so many message headers. The number of headers listed depends on the speed of your terminal. This can be overridden by specifying the number of headers you want with the *window* option. *Mail* maintains a notion of the current "window" into your messages for the purposes of printing headers. Use the **z** command to move forward and back a window. You can move *Mail's* notion of the current window directly to a particular message by using, for example,

```
headers 40
```

to move *Mail's* attention to the messages around message 40. The **headers** command can be abbreviated to **h**.

help Print a brief and usually out of date help message about the commands in *Mail*. Refer to this manual instead.

hold Arrange to hold a list of messages in the system mailbox, instead of moving them to the file *mbox* in your home directory. If you set the binary option *hold*, this will happen by default.

if Commands in your ".mailrc" file can be executed conditionally depending on whether you are sending or receiving mail with the **if** command. For example, you can do:

```
if receive
        commands...
endif
```

An **else** form is also available:

```
if send
        commands...
else
```

commands...
endif

Note that the only allowed conditions are **receive** and **send**.

ignore
Add the list of header fields named to the *ignore list.* Header fields in the ignore list are not printed on your terminal when you print a message. This allows you to suppress printing of certain machine-generated header fields, such as *Via* which are not usually of interest. The **Type** and **Print** commands can be used to print a message in its entirety, including ignored fields. If **ignore** is executed with no arguments, it lists the current set of ignored fields.

list List the vaild *Mail* commands.

mail Send mail to one or more people. If you have the *ask* option set, *Mail* will prompt you for a subject to your message. Then you can type in your message, using tilde escapes as described in section 4 to edit, print, or modify your message. To signal your satisfaction with the message and send it, type control-d at the beginning of a line, or a . alone on a line if you set the option *dot.* To abort the message, type two interrupt characters (RUBOUT by default) in a row or use the ~q escape.

mbox
Indicate that a list of messages be sent to *mbox* in your home directory when you quit. This is the default action for messages if you do *not* have the *hold* option set.

next The **next** command goes to the next message and types it. If given a message list, **next** goes to the first such message and types it. Thus,

next root

goes to the next message sent by "root" and types it. The **next** command can be abbreviated to simply a newline, which means that one can go to and type a message by simply giving its message number or one of the magic characters "↑" "." or "$". Thus,

prints the current message and

4

prints message 4, as described previously.

preserve
Same as **hold**. Cause a list of messages to be held in your system mailbox when you quit.

quit Leave *Mail* and update the file, folder, or system mailbox your were reading. Messages that you have examined are marked as "read" and messages that existed when you started are marked as "old." If you were editing your system mailbox and if you have set the binary option *hold*, all messages which have not been deleted, saved, or mboxed will be retained in your system mailbox. If you were editing your system mailbox and you did *not* have *hold* set, all messages which have not been deleted, saved, or preserved will be moved to the file *mbox* in your home directory.

reply Frame a reply to a single message. The reply will be sent to the person who sent you the message to which you are replying, plus all the people who received the original message, except you. You can add people using the ~t and ~c tilde escapes. The subject in your reply is formed by prefacing the subject in the original message with "Re:" unless it already began thus. If the original message included a "reply-to" header field, the reply will go *only* to the recipient named by "reply-to." You type in your message using the same conventions available to you through the **mail** command.

save It is often useful to be able to save messages on related topics in a file. The **save** command gives you ability to do this. The **save** command takes as argument a lit of message numbers, followed by the name of the file on which to save the messages. The messages are appended to the named file, thus allowing one to keep several messages in the file, stored in the order they were put there. The **save** command can be abbreviated to **s**. An example of the **save** command relative to our running example is:

```
s 1 2 tuitionmail
```

Saved messages are not automatically saved in *mbox* at quit time, nor are they selected by the **next** command described above, unless explicitly specified.

set Set an option or give an option a value. Used to customize *Mail.* Section 5.3 contains a list of the options. Options can be *binary*, in which case they are *on* or *off*, or *valued.* To set a binary option *option on*, do

```
set option
```

To give the valued option *option* the value *value*, do

```
set option=value
```

Several options can be specified in a single **set** command.

shell The **shell** command allows you to escape to the shell. **Shell** invokes an interactive shell and allows you to type commands to it. When you leave the shell, you will return to *Mail.* The shell used is a default assumed by *Mail*; you can override this default by setting the valued option "SHELL," eg:

```
set SHELL=/bin/csh
```

source
The **source** command reads *Mail* commands from a file. It is useful when you are trying to fix your ".mailrc" file and you need to re-read it.

top The **top** command takes a message list and prints the first five lines of each addressed message. It may be abbreviated to **to**. If you wish, you can change the number of lines that **top** prints out by setting the valued option "toplines." On a CRT terminal,

```
set toplines=10
```

might be preferred.

type Print a list of messages on your terminal. If you have set the option *crt* to a number and the total number of lines in the messages you are printing exceed that specified by *crt*, the messages will be printed by a terminal paging program such as *more.*

undelete
The **undelete** command causes a message that had been deleted previously to regain its initial status. Only messages that have been deleted may be undeleted. This command may be abbreviated to **u**.

unset
Reverse the action of setting a binary or valued option.

visual
It is often useful to be able to invoke one of two editors, based on the type of terminal one is using. To invoke a display oriented editor, you can use the **visual** command. The operation of the **visual** command is otherwise identical to that of the **edit** command.

Both the **edit** and **visual** commands assume some default text editors. These default editors can be overridden by the valued options "EDITOR" and "VISUAL" for the standard and screen editors. You might want to do:

```
set EDITOR=/usr/ucb/ex VISUAL=/usr/ucb/vi
```

write The **save** command always writes the entire message, including the headers, into the file. If you want to write just the message itself, you can use the **write** command. The **write** command has the same syntax as the **save** command, and can be abbreviated to simply **w**. Thus, we could write the second message by doing:

```
w 2 file.c
```

As suggested by this example, the **write** command is useful for such tasks as sending and receiving source program text over the message system.

z *Mail* presents message headers in windowfuls as described under the **headers** command. You can move *Mail's* attention forward to the next window by giving the

```
z+
```

command. Analogously, you can move to the previous window with:

```
z−
```

5.3. Custom options

Throughout this manual, we have seen examples of binary and valued options. This section describes each of the options in alphabetical order, including some that you have not seen yet. To avoid confusion, please note that the options are either all lower case letters or all upper case letters. When I start a sentence such as: "Ask" causes *Mail* to prompt you for a subject header, I am only capitalizing "ask" as a courtesy to English.

EDITOR
The valued option "EDITOR" defines the pathname of the text editor to be used in the **edit** command and ~e. If not defined, a standard editor is used.

SHELL
The valued option "SHELL" gives the path name of your shell. This shell is used for the ! command and ~! escape. In addition, this shell expands file names with shell metacharacters like * and ? in them.

VISUAL
The valued option "VISUAL" defines the pathname of your screen editor for use in the **visual** command and ~v escape. A standard screen editor is used if you do not define one.

append
The "append" option is binary and causes messages saved in *mbox* to be appended to the end rather than prepended. Normally, *Mail*will *mbox* in the same order that the system puts messages in your system mailbox. By setting "append," you are requesting that *mbox* be appended to regardless. It is in any event quicker to append.

ask "Ask" is a binary option which causes *Mail* to prompt you for the subject of each message you send. If you respond with simply a newline, no subject field will be sent.

askcc
"Askcc" is a binary option which causes you to be prompted for additional carbon copy recipients at the end of each message. Responding with a newline shows your satisfaction with the current list.

autoprint
"Autoprint" is a binary option which causes the **delete** command to behave like **dp** — thus, after deleting a message, the next one will be typed automatically. This is useful to quickly scanning and deleting messages in your mailbox.

debug
The binary option "debug" causes debugging information to be displayed. Use of this option is the same as useing the

−d command line flag.

dot "Dot" is a binary option which, if set, causes *Mail* to interpret a period alone on a line as the terminator of a message you are sending.

escape
To allow you to change the escape character used when sending mail, you can set the valued option "escape." Only the first character of the "escape" option is used, and it

must be doubled if it is to appear as the first character of a line of your message. If you change your escape character, then ~ loses all its special meaning, and need no longer be doubled at the beginning of a line.

folder
The name of the directory to use for storing folders of messages. If this name begins with a '/' *Mail* considers it to be an absolute pathname; otherwise, the folder directory is found relative to your home directory.

hold The binary option "hold" causes messages that have been read but not manually dealt with to be held in the system mailbox. This prevents such messages from being automatically swept into your mbox.

ignore
The binary option "ignore" causes RUBOUT characters from your terminal to be ignored and echoed as @'s while you are sending mail. RUBOUT characters retain their original meaning in *Mail* command mode. Setting the "ignore" option is equivalent to supplying the **−i** flag on the command line as described in section 6.

ignoreeof
An option related to "dot" is "ignoreeof" which makes *Mail* refuse to accept a control−d as the end of a message. "Ignoreeof" also applies to *Mail* command mode.

keep The "keep" option causes *Mail* to truncate your system mailbox instead of deleting it when it is empty. This is useful if you elect to protect your mailbox, which you would do with the shell command:

```
chmod 600 /usr/spool/mail/yourname
```

where *yourname* is your login name. If you do not do this, anyone can probably read your mail, although people usually don't.

keepsave
When you save a message, *Mail* usually discards it when you **quit**. To retain all saved messages, set the "keepsave" option.

metoo
When sending mail to an alias, *Mail* makes sure that if you are included in the alias, that mail will not be sent to you. This is useful if a single alias is being used by all members of the group. If however, you wish to receive a copy of all the messages you send to the alias, you can set the binary option "metoo."

noheader
The binary option "noheader" suppresses the printing of the version and headers when *Mail* is first invoked. Setting this option is the same as using −N on the command line.

nosave
Normally, when you abort a message with two RUBOUTs, *Mail* copies the partial letter to the file "dead.letter" in your home directory. Setting the binary option "nosave" prevents this.

quiet The binary option "quiet" suppresses the printing of the version when *Mail* is first invoked, as well as printing the for example "Message 4:" from the **type** command.

record
If you love to keep records, then the valued option "record" can be set to the name of a file to save your outgoing mail. Each new message you send is appended to the end of the file.

screen
When *Mail* initially prints the message headers, it determines the number to print by looking at the speed of your terminal. The faster your terminal, the more it prints. The valued option "screen" overrides this calculation and specifies how many message headers you want printed. This number is also used for scrolling with the **z** command.

sendmail

To alternate delivery system, set the ‘‘sendmail’’ option to the full pathname of the program to use. Note: this is not for everyone! Most people should use the default delivery system.

toplines

The valued option ‘‘toplines’’ defines the number of lines that the ‘‘top’’ command will print out instead of the default five lines.

verbose

The binary option "verbose" causes *Mail* to invoke sendmail with the −v flag, which causes it to go into versbose mode and announce expansion of aliases, etc. Setting the "verbose" option is equivalent to invoking *Mail* with the −v flag as described in section 6.

6. Command line options

This section describes command line options for *Mail* and what they are used for.

−N Suppress the initial printing of headers.

−d Turn on debugging information. Not of general interest.

−f file
Show the messages in *file* instead of your system mailbox. If *file* is omitted, *Mail* reads *mbox* in your home directory.

−i Ignore tty interrupt signals. Useful on noisy phone lines, which generate spurious RUBOUT or DELETE characters. It's usually more effective to change your interrupt character to control−c, for which see the *stty* shell command.

−n Inhibit reading of /usr/lib/Mail.rc. Not generally useful, since /usr/lib/Mail.rc is usually empty.

−s string
Used for sending mail. *String* is used as the subject of the message being composed. If *string* contains blanks, you must surround it with quote marks.

−u name
Read *names's* mail instead of your own. Unwitting others often neglect to protect their mailboxes, but discretion is advised. Essentially, **−u user** is a shorthand way of doing **−f /usr/spool/user**.

−v Use the **−v** flag when invoking sendmail. This feature may also be enabled by setting the the option "verbose".

The following command line flags are also recognized, but are intended for use by programs invoking *Mail* and not for people.

−T file
Arrange to print on *file* the contents of the *article-id* fields of all messages that were either read or deleted. **−T** is for the *readnews* program and should NOT be used for reading your mail.

−h number
Pass on hop count information. *Mail* will take the number, increment it, and pass it with **−h** to the mail delivery system. **−h** only has effect when sending mail and is used for network mail forwarding.

−r name
Used for network mail forwarding: interpret *name* as the sender of the message. The *name* and **−r** are simply sent along to the mail delivery system. Also, *Mail* will wait for the message to be sent and return the exit status. Also restricts formatting of message.

Note that **−h** and **−r**, which are for network mail forwarding, are not used in practice since mail forwarding is now handled separately. They may disappear soon.

7. Format of messages

This section describes the format of messages. Messages begin with a *from* line, which consists of the word "From" followed by a user name, followed by anything, followed by a date in the format returned by the *ctime* library routine described in section 3 of the Unix Programmer's Manual. A possible *ctime* format date is:

```
Tue Dec  1 10:58:23 1981
```

The *ctime* date may be optionally followed by a single space and a time zone indication, which should be three capital letters, such as PDT.

Following the *from* line are zero or more *header field* lines. Each header field line is of the form:

```
name: information
```

Name can be anything, but only certain header fields are recognized as having any meaning. The recognized header fields are: *article-id*, *bcc*, *cc*, *from*, *reply-to*, *sender*, *subject*, and *to*. Other header fields are also significant to other systems; see, for example, the current Arpanet message standard for much more on this topic. A header field can be continued onto following lines by making the first character on the following line a space or tab character.

If any headers are present, they must be followed by a blank line. The part that follows is called the *body* of the message, and must be ASCII text, not containing null characters. Each line in the message body must be terminated with an ASCII newline character and no line may be longer than 512 characters. If binary data must be passed through the mail system, it is suggested that this data be encoded in a system which encodes six bits into a printable character. For example, one could use the upper and lower case letters, the digits, and the characters comma and period to make up the 64 characters. Then, one can send a 16-bit binary number as three characters. These characters should be packed into lines, preferably lines about 70 characters long as long lines are transmitted more efficiently.

The message delivery system always adds a blank line to the end of each message. This blank line must not be deleted.

The UUCP message delivery system sometimes adds a blank line to the end of a message each time it is forwarded through a machine.

It should be noted that some network transport protocols enforce limits to the lengths of messages.

8. Glossary

This section contains the definitions of a few phrases peculiar to *Mail.*

alias An alternative name for a person or list of people.

flag An option, given on the command line of *Mail,* prefaced with a **−**. For example, **−f** is a flag.

header field
At the beginning of a message, a line which contains information that is part of the structure of the message. Popular header fields include *to*, *cc*, and *subject.*

mail A collection of messages. Often used in the phrase, "Have you read your mail?"

mailbox
The place where your mail is stored, typically in the directory /usr/spool/mail.

message
A single letter from someone, initially stored in your *mailbox.*

message list
A string used in *Mail* command mode to describe a sequence of messages.

option
A piece of special purpose information used to tailor *Mail* to your taste. Options are specified with the **set** command.

9. Summary of commands, options, and escapes

This section gives a quick summary of the *Mail* commands, binary and valued options, and tilde escapes.

The following table describes the commands:

Command	*Description*
!	Single command escape to shell
-	Back up to previous message
Print	Type message with ignored fields
Reply	Reply to author of message only
Type	Type message with ignored fields
alias	Define an alias as a set of user names
alternates	List other names you are known by
chdir	Change working directory, home by default
copy	Copy a message to a file or folder
delete	Delete a list of messages
dt	Delete current message, type next message
endif	End of conditional statement; see **if**
edit	Edit a list of messages
else	Start of else part of conditional; see **if**
exit	Leave mail without changing anything
file	Interrogate/change current mail file
folder	Same as **file**
folders	List the folders in your folder directory
from	List headers of a list of messages
headers	List current window of messages
help	Print brief summary of *Mail* commands
hold	Same as **preserve**
if	Conditional execution of *Mail* commands
ignore	Set/examine list of ignored header fields
list	List valid *Mail* commands
local	List other names for the local host
mail	Send mail to specified names
mbox	Arrange to save a list of messages in *mbox*
next	Go to next message and type it
preserve	Arrange to leave list of messages in system mailbox
quit	Leave *Mail*; update system mailbox, *mbox* as appropriate
reply	Compose a reply to a message
save	Append messages, headers included, on a file
set	Set binary or valued options
shell	Invoke an interactive shell
top	Print first so many (5 by default) lines of list of messages
type	Print messages
undelete	Undelete list of messages
unset	Undo the operation of a set
visual	Invoke visual editor on a list of messages
write	Append messages to a file, don't include headers
z	Scroll to next/previous screenful of headers

The following table describes the options. Each option is shown as being either a binary or valued option.

Option	*Type*	*Description*
EDITOR	*valued*	Pathname of editor for ~e and **edit**
SHELL	*valued*	Pathname of shell for **shell**, ~! and **!**
VISUAL	*valued*	Pathname of screen editor for ~v, **visual**
append	*binary*	Always append messages to end of *mbox*
ask	*binary*	Prompt user for Subject: field when sending
askcc	*binary*	Prompt user for additional Cc's at end of message
autoprint	*binary*	Print next message after **delete**
crt	*valued*	Minimum number of lines before using *more*
debug	*binary*	Print out debugging information
dot	*binary*	Accept . alone on line to terminate message input
escape	*valued*	Escape character to be used instead of ~
folder	*valued*	Directory to store folders in
hold	*binary*	Hold messages in system mailbox by default
ignore	*binary*	Ignore RUBOUT while sending mail
ignoreeof	*binary*	Don't terminate letters/command input with ↑D
keep	*binary*	Don't unlink system mailbox when empty
keepsave	*binary*	Don't delete saved messages by default
metoo	*binary*	Include sending user in aliases
noheader	*binary*	Suppress initial printing of version and headers
nosave	*binary*	Don't save partial letter in *dead.letter*
quiet	*binary*	Suppress printing of *Mail* version and message numbers
record	*valued*	File to save all outgoing mail in
screen	*valued*	Size of window of message headers for **z**, etc.
sendmail	*valued*	Choose alternate mail delivery system
toplines	*valued*	Number of lines to print in **top**
verbose	*binary*	Invoke sendmail with the **−v** flag

The following table summarizes the tilde escapes available while sending mail.

Escape	*Arguments*	*Description*
~!	*command*	Execute shell command
~c	*name ...*	Add names to Cc: field
~d		Read *dead.letter* into message
~e		Invoke text editor on partial message
~f	*messages*	Read named messages
~h		Edit the header fields
~m	*messages*	Read named messages, right shift by tab
~p		Print message entered so far
~q		Abort entry of letter; like RUBOUT
~r	*filename*	Read file into message
~s	*string*	Set Subject: field to *string*
~t	*name ...*	Add names to To: field
~v		Invoke screen editor on message
~w	*filename*	Write message on file
~|	*command*	Pipe message through *command*
~~	*string*	Quote a ~ in front of *string*

The following table shows the command line flags that *Mail* accepts:

Flag	Description
−N	Suppress the initial printing of headers
−T *file*	Article-id's of read/deleted messages to *file*
−d	Turn on debugging
−f *file*	Show messages in *file* or *~/mbox*
−h *number*	Pass on hop count for mail forwarding
−i	Ignore tty interrupt signals
−n	Inhibit reading of /usr/lib/Mail.rc
−r *name*	Pass on *name* for mail forwarding
−s *string*	Use *string* as subject in outgoing mail
−u *name*	Read *name's* mail instead of your own
−v	Invoke sendmail with the −v flag

Notes: **−T**, **−d**, **−h**, and **−r** are not for human use.

10. Conclusion

Mail is an attempt to provide a simple user interface to a variety of underlying message systems. Thanks are due to the many users who contributed ideas and testing to *Mail*.

DC — An Interactive Desk Calculator

Robert Morris

Lorinda Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

DC is an interactive desk calculator program implemented on the UNIX† time-sharing system to do arbitrary-precision integer arithmetic. It has provision for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The size of numbers that can be manipulated is limited only by available core storage. On typical implementations of UNIX, the size of numbers that can be handled varies from several hundred digits on the smallest systems to several thousand on the largest.

November 15, 1978

†UNIX is a Trademark of Bell Laboratories.

DC − An Interactive Desk Calculator

Robert Morris

Lorinda Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

DC is an arbitrary precision arithmetic package implemented on the UNIX† time-sharing system in the form of an interactive desk calculator. It works like a stacking calculator using reverse Polish notation. Ordinarily DC operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained.

A language called BC [1] has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by DC. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into DC are put on a push-down stack. DC commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

SYNOPTIC DESCRIPTION

Here we describe the DC commands that are intended for use by people. The additional commands that are intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

number

The value of the number is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A−F which are treated as digits with values 10−15 respectively. The number may be preceded by an underscore to input a negative number. Numbers may contain decimal points.

+ − * % ^

The top two values on the stack are added (+), subtracted (−), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. See the detailed description below for the treatment of numbers with decimal points. An exponent must not have any digits after the decimal point.

†UNIX is a Trademark of Bell Laboratories.

s*x*

The top of the main stack is popped and stored into a register named *x*, where *x* may be any character. If the s is capitalized, *x* is treated as a stack and the value is pushed onto it. Any character, even blank or new-line, is a valid register name.

l*x*

The value in register *x* is pushed onto the stack. The register *x* is not altered. If the l is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

All registers start with empty value which is treated as a zero by the command l and is treated as an error by the command L.

d

The top value on the stack is duplicated.

p

The top value on the stack is printed. The top value remains unchanged.

f

All values on the stack and in registers are printed.

x

treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of DC commands.

[...]

puts the bracketed character string onto the top of the stack.

q

exits the program. If executing a string, the recursion level is popped by two. If q is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

<*x* **>***x* **=***x* **!<***x* **!>***x* **!=***x*

The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation. Exclamation point is negation.

v

replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer. For the treatment of numbers with decimal points, see the detailed description below.

!

interprets the rest of the line as a UNIX command. Control returns to DC when the UNIX command terminates.

c

All values on the stack are popped; the stack becomes empty.

i

The top value on the stack is popped and used as the number radix for further input. If **i** is capitalized, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o

The top value on the stack is popped and used as the number radix for further output. If **o** is capitalized, the value of the output base is pushed onto the stack.

k

The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If **k** is capitalized, the value of the scale factor is pushed onto the stack.

z

The value of the stack level is pushed onto the stack.

?

A line of input is taken from the input source (usually the console) and executed.

DETAILED DESCRIPTION

Internal Representation of Numbers

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0−99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100's complement notation, which is analogous to two's complement notation for binary numbers. The high order digit of a negative number is always −1 and all other digits are in the range 0−99. The digit preceding the high order −1 digit is never a 99. The representation of −157 is 43,98,−1. We shall call this the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when that is convenient, as it sometimes is.

An additional byte is stored with each number beyond the high order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,*3* where the scale has been italicized to emphasize the fact that it is not the high order digit. The value of this extra byte is called the **scale factor** of the number.

The Allocator

DC uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is done through the allocator. Associated with each string in the allocator is a four-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and DC is done via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of 2. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Left-over strings are put on the free list. If there are no larger strings, the allocator tries to coalesce smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long. This is an implementation of the 'buddy system' of allocation described in [2].

Failing to find a string of the proper length after coalescing, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in DC. If at any time in the process of trying to allocate a string, the allocator runs out of headers, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward-spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end-of-string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

Internal Arithmetic

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations. **scale** is the bound on the number of decimal places retained in arithmetic computations. **scale** may be set to the number on the top of the stack truncated to an integer with the **k** command. **K** may be used to push the value of **scale** on the stack. **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations will include the exact effect of **scale** on the computations.

Addition and Subtraction

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

Finally, the addition is performed digit by digit from the low order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers replacing the high-order configuration $99,-1$ by the digit -1. In any case, digits which are not in the range $0-99$ must be brought into that range, propagating any carries or borrows that result.

Multiplication

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly mimics the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register scale and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

Division

The scales are removed from the two operands. Zeros are appended or digits removed from the dividend to make the scale of the result of the integer division equal to the internal quantity scale. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. It may turn out be one unit too low, but if it is, the next trial quotient will be larger than 99 and this will be adjusted at the end of the process. The trial digit is multiplied by the divisor and the result subtracted from the dividend and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form, with propagation of carry as needed. The sign is set from the sign of the operands.

Remainder

The division routine is called and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

Square Root

The scale is stripped from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity scale and the scale of the operand.

The method used to compute sqrt(y) is Newton's method with successive approximations by the rule

$$x_{n+1} = \tfrac{1}{2}(x_n + \frac{y}{x_n})$$

The initial guess is found by taking the integer square root of the top two digits.

Exponentiation

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive and the base is divided into one. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

Input Conversion and Base

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with a _. The hexadecimal digits A−F correspond to the numbers 10−15 regardless of input base. The **i** command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base is initialized to 10 but may, for example be changed to 8 or 16 to do octal or hexadecimal to decimal conversions. The command **I** will push the value of the input base on the stack.

Output Commands

The command **p** causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers can be output by typing the command **f**. The **o** command can be used to change the output base. This command uses the top of the stack, truncated to an integer as the base for all further output. The output base in initialized to 10. It will work correctly for any base. The command **O** pushes the value of the output base on the stack.

Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a \ indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 can be used for decimal-octal or decimal-hexadecimal conversions.

Internal Registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands **s** and **l**. The command **s***x* pops the top of the stack and stores the result in register **x**. *x* can be any character. **l***x* puts the contents of register **x** on the top of the stack. The **l** command has no effect on the contents of register *x*. The **s** command, however, is destructive.

Stack Commands

The command **c** clears the stack. The command **d** pushes a duplicate of the number on the top of the stack on the stack. The command **z** pushes the stack size on the stack. The command **X** replaces the number on the top of the stack with its scale factor. The command **Z** replaces the top of the stack with its length.

Subroutine Definitions and Calls

Enclosing a string in **[]** pushes the ascii string on the stack. The **q** command quits or in executing a string, pops the recursion levels by two.

Internal Registers − Programming DC

The load and store commands together with **[]** to store strings, **x** to execute and the testing commands '<', '>', '=', '!<', '!>', '!=' can be used to program DC. The **x** command assumes the top of the stack is an string of DC commands and executes it. The testing commands compare the top two elements on the stack and if the relation holds, execute the register that follows the relation. For example, to print the numbers 0-9,

```
[lipl+ si li10>a]sa
0si lax
```

Push-Down Registers and Arrays

These commands were designed for used by a compiler, not by people. They involve push-down registers and arrays. In addition to the stack that commands work on, DC can be thought of as having individual stacks for each register. These registers are operated on by the commands S and L. S*x* pushes the top value of the main stack onto the stack for the register *x*. L*x* pops the stack for register *x* and puts the result on the main stack. The commands s and l also work on registers but not as push-down stacks. l doesn't effect the top of the register stack, and s destroys what was there before.

The commands to work on arrays are : and ;. :*x* pops the stack and uses this value as an index into the array *x*. The next element on the stack is stored at this index in *x*. An index must be greater than or equal to 0 and less than 2048. ;*x* is the command to load the main stack from the array *x*. The value on the top of the stack is the index into the array *x* of the value to be loaded.

Miscellaneous Commands

The command ! interprets the rest of the line as a UNIX
command and passes it to UNIX to execute. One other compiler command is Q. This command uses the top of the stack as the number of levels of recursion to skip.

DESIGN CHOICES

The real reason for the use of a dynamic storage allocator was that a general purpose program could be (and in fact has been) used for a variety of other tasks. The allocator has some value for input and for compiling (i.e. the bracket [...] commands) where it cannot be known in advance how long a string will be. The result was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25% of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of hardware limitations and at the cost of 5% in space, debugging was made a great deal easier and decimal output was made much faster.

The reason for a stack-type arithmetic design was to permit all DC commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases was to provide an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation which had global notions of scale and base did not work out well. If the value of scale were to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations. The value of scale is not used for any essential purpose by any part of the program and it is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The design rationale for the choices for the scales of the results of arithmetic were that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give him the result 5.017 without requiring him to unnecessarily specify his rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands and it seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for **scale**. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places and there is simply no way to guess how many places the user wants. In this case only, the user must specify a **scale** to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

References

[1] L. L. Cherry, R. Morris, *BC — An Arbitrary Precision Desk-Calculator Language.*

[2] K. C. Knowlton, *A Fast Storage Allocator,* Comm. ACM **8**, pp. 623-625 (Oct. 1965).

BC – An Arbitrary Precision Desk-Calculator Language

Lorinda Cherry

Robert Morris

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

BC is a language and a compiler for doing arbitrary precision arithmetic on the PDP-11 under the UNIX† time-sharing system. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers.

These routines are themselves based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution.

Two five hundred-digit numbers can be multiplied to give a thousand digit result in about ten seconds.

A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Some of the uses of this compiler are

– to do computation with large integers,

– to do computation accurate to many decimal places,

– conversion of numbers from one base to another base.

November 12, 1978

†UNIX is a Trademark of Bell Laboratories.

BC – An Arbitrary Precision Desk-Calculator Language

Lorinda Cherry

Robert Morris

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

BC is a language and a compiler for doing arbitrary precision arithmetic on the UNIX† time-sharing system [1]. The compiler was written to make conveniently available a collection of routines (called DC [5]) which are capable of doing arithmetic on integers of arbitrary size. The compiler is by no means intended to provide a complete programming language. It is a minimal language facility.

There is a scaling provision that permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of UNIX.

The syntax of BC has been deliberately selected to agree substantially with the C language [2]. Those who are familiar with C will find few surprises in this language.

Simple Computations with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

```
142857 + 285714
```

the program responds immediately with the line

```
428571
```

The operators −, *, /, %, and ^ can also be used; they indicate subtraction, multiplication, division, remaindering, and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the 'unary' minus sign). The expression

```
7+−3
```

is interpreted to mean that −3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in Fortran, with ^ having the greatest binding power, then * and % and /, and finally + and −. Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right. The two expressions

†UNIX is a Trademark of Bell Laboratories.

```
a^b^c  and  a^(b^c)
```

are equivalent, as are the two expressions

```
a*b*c  and  (a*b)*c
```

BC shares with Fortran and C the undesirable convention that

```
a/b*c  is equivalent to  (a/b)*c
```

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

has the effect of increasing by three the value of the contents of the register named x. When, as in this case, the outermost operator is an =, the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (but see scaling below). The lines

```
x = sqrt(191)
x
```

produce the printed result

```
13
```

Bases

There are special internal quantities, called 'ibase' and 'obase'. The contents of 'ibase', initially set to 10, determines the base used for interpreting numbers read in. For example, the lines

```
ibase = 8
11
```

will produce the output line

```
9
```

and you are all set up to do octal to decimal conversions. Beware, however of trying to change the input base back to decimal by typing

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement will have no effect. For those who deal in hexadecimal notation, the characters A−F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10−15 respectively. The statement

```
ibase = A
```

will change you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of 'obase', initially set to 10, are used as the base for output numbers. The lines

```
obase = 16
1000
```

will produce the output line

```
3E8
```

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted, and they are sometimes useful. For example, large numbers can be output in groups of five digits by setting 'obase' to 100000. Strange (i.e. 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with \. Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

It is best to remember that 'ibase' and 'obase' have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

Scaling

A third special internal quantity called 'scale' is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules. For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result. For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity 'scale'. The scale of a quotient is the contents of the internal quantity 'scale'. The scale of a remainder is the sum of the scales of the quotient and the divisor. The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer. The scale of a square root is set to the maximum of the scale of the argument and the contents of 'scale'.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The contents of 'scale' must be no greater than 99 and no less than 0. It is initially set to 0. In case you need more than 99 fraction digits, you may arrange your own scaling.

The internal quantities 'scale', 'ibase', and 'obase' can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of 'scale' by one, and the line

```
scale
```

causes the current value of 'scale' to be printed.

The value of 'scale' retains its meaning as a number of decimal digits to be retained in internal computation even when 'ibase' or 'obase' are not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

Functions

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace }. Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms

```
return
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one 'auto' statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
        auto z
        z = x*y
        return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: b().

If the function *a* above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed and the line

```
x = a(a(3,4),5)
```

would cause the value of x to become 60.

Subscripted Variables

A single lower-case letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names which refer to whole arrays cannot be used in any other contexts.

Control Statements

The 'if', the 'while', and the 'for' statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the six relational operators <, >, <=, >=, ==, or !=. The relation == stands for 'equal to' and != stands for 'not equal to'. The meaning of the remaining relational operators is clear.

BEWARE of using = instead of == in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but = really will not do a comparison.

The 'if' statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The 'while' statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The 'for' statement begins by executing 'expression1'. Then the relation is tested and, if true, the statements in the range of the 'for' are executed. Then 'expression2' is executed. The relation is tested, and so on. The typical use of the 'for' statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here are some examples of the use of the control statements.

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The line

```
        f(a)
```

will print *a* factorial if *a* is a positive integer. Here is the definition of a function which will compute values of the binomial coefficient (m and n are assumed to be positive integers).

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
        auto a, b, c, d, n
        a = 1
        b = 1
        c = 1
        d = 0
        n = 1
        while(1==1){
                a = a*x
                b = b*n
                c = c + a/b
                n = n + 1
                if(c==d) return(c)
                d = c
        }
}
```

Some Details

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

causes a value to be assigned to x and also increments i before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language. Consult the appendix or the C manuals [2] for their exact workings.

x=y=z	is the same as	x=(y=z)
x =+ y		x = x+y
x =− y		x = x−y
x =* y		x = x*y
x =/ y		x = x/y
x =% y		x = x%y
x =^ y		x = x^y
x++		(x=x+1)−1
x−−		(x=x−1)+1
++x		x = x+1
−−x		x = x−1

Even if you don't intend to use the constructs, if you type one inadvertently, something correct but unexpected may happen.

WARNING! In some of these constructions, spaces are significant. There is a real difference between x=−y and x= −y. The first replaces x by x−y and the second by −y.

Three Important Things

1. To exit a BC program, type 'quit'.

2. There is a comment convention identical to that of C and of PL/I. Comments begin with '/*' and end with '*/'.

3. There is a library of math functions which may be obtained by typing at command level

```
bc −l
```

This command will load a set of library functions which, at the time of writing, consists of sine (named 's'), cosine ('c'), arctangent ('a'), natural logarithm ('l'), exponential ('e') and Bessel functions of integer order ('j(n,x)'). Doubtless more functions will be added in time. The library sets the scale to 20. You can reset it to something else if you like. The design of these mathematical library routines is discussed elsewhere [3].

If you type

```
bc file ...
```

BC will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

Acknowledgement

The compiler is written in YACC [4]; its original version was written by S. C. Johnson.

References

[1] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Bell Laboratories, 1978.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[3] R. Morris, *A Library of Reference Standard Mathematical Subroutines*, Bell Laboratories internal memorandum, 1975.

[4] S. C. Johnson, *YACC — Yet Another Compiler-Compiler*. Bell Laboratories Computing Science Technical Report #32, 1978.

[5] R. Morris and L. L. Cherry, *DC — An Interactive Desk Calculator*.

Appendix

1. Notation

In the following pages syntactic categories are in *italics*; literals are in **bold**; material in brackets [ ] is optional.

2. Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

2.1. Comments

Comments are introduced by the characters /* and terminated by */.

2.2. Identifiers

There are three kinds of identifiers — ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower-case letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict; a program can have a variable named **x**, an array named **x** and a function named **x**, all of which are separate and distinct.

2.3. Keywords

The following are reserved keywords:

ibase	**if**
obase	**break**
scale	**define**
sqrt	**auto**
length	**return**
while	**quit**
for	

2.4. Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits **A**—**F** are also recognized as digits with values 10—15, respectively.

3. Expressions

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

3.1. Primitive expressions

3.1.1. Named expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

3.1.1.1. *identifiers*

Simple identifiers are named expressions. They have an initial value of zero.

3.1.1.2. *array-name* [*expression*]

Array elements are named expressions. They have an initial value of zero.

3.1.1.3. scale, ibase and obase

The internal registers **scale**, **ibase** and **obase** are all named expressions. **scale** is the number of digits after the decimal point to be retained in arithmetic operations. **scale** has an initial value of zero. **ibase** and **obase** are the input and output number radix respectively. Both **ibase** and **obase** have initial values of 10.

3.1.2. Function calls

3.1.2.1. *function-name* ([*expression* [,*expression*. . .]])

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is zero if no expression is provided or if there is no return statement.

3.1.2.2. sqrt (*expression*)

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale**, whichever is larger.

3.1.2.3. length (*expression*)

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

3.1.2.4. scale (*expression*)

The result is the scale of the expression. The scale of the result is zero.

3.1.3. Constants

Constants are primitive expressions.

3.1.4. Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

3.2. Unary operators

The unary operators bind right to left.

3.2.1. − *expression*

The result is the negative of the expression.

3.2.2. + + *named-expression*

The named expression is incremented by one. The result is the value of the named expression after incrementing.

3.2.3. − − *named-expression*

The named expression is decremented by one. The result is the value of the named expression after decrementing.

3.2.4. *named-expression* + +

The named expression is incremented by one. The result is the value of the named expression before incrementing.

3.2.5. *named-expression* − −

The named expression is decremented by one. The result is the value of the named expression before decrementing.

3.3. Exponentiation operator

The exponentiation operator binds right to left.

3.3.1. *expression* ^ *expression*

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If a is the scale of the left expression and b is the absolute value of the right expression, then the scale of the result is:

min ($a \times b$, max (**scale**, a))

3.4. Multiplicative operators

The operators *, /, % bind left to right.

3.4.1. *expression* * *expression*

The result is the product of the two expressions. If a and b are the scales of the two expressions, then the scale of the result is:

min ($a + b$, max (**scale**, a, b))

3.4.2. *expression* / *expression*

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

3.4.3. *expression* % *expression*

The % operator produces the remainder of the division of the two expressions. More precisely, $a\%b$ is $a - a/b*b$.

The scale of the result is the sum of the scale of the divisor and the value of **scale**

3.5. Additive operators

The additive operators bind left to right.

3.5.1. *expression* + *expression*

The result is the sum of the two expressions. The scale of the result is the maximun of the scales of the expressions.

3.5.2. *expression* − *expression*

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

3.6. assignment operators

The assignment operators bind right to left.

3.6.1. *named-expression* = *expression*

This expression results in assigning the value of the expression on the right to the named expression on the left.

3.6.2. *named-expression* =+ *expression*

3.6.3. *named-expression* =− *expression*

3.6.4. *named-expression* =* *expression*

3.6.5. *named-expression* =/ *expression*

3.6.6. *named-expression* =% *expression*

3.6.7. *named-expression* =^ *expression*

The result of the above expressions is equivalent to "named expression = named expression OP expression", where OP is the operator after the = sign.

4. Relations

Unlike all other operators, the relational operators are only valid as the object of an **if**, **while**, or inside a **for** statement.

4.1. *expression* < *expression*

4.2. *expression* > *expression*

4.3. *expression* <= *expression*

4.4. *expression* >= *expression*

4.5. *expression* == *expression*

4.6. *expression* != *expression*

5. Storage classes

There are only two storage classes in BC, global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in either C or PL/I. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

6. Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

6.1. Expression statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

6.2. Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

6.3. Quoted string statements

"any string"

This statement prints the string inside the quotes.

6.4. If statements

if (*relation*) *statement*

The substatement is executed if the relation is true.

6.5. While statements

while (*relation*) *statement*

The statement is executed while the relation is true. The test occurs before each execution of the statement.

6.6. For statements

for (*expression*; *relation*; *expression*) *statement*

The for statement is the same as

```
first-expression
while (relation ) {
        statement
        last-expression
}
```

All three expressions must be present.

6.7. Break statements

break

break causes termination of a **for** or **while** statement.

6.8. Auto statements

auto *identifier* [,*identifier*]

The auto statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition.

6.9. Define statements

define([*parameter* [,*parameter*. . .]] **) {**
    statements **}**

The define statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

6.10. Return statements

return

return(*expression* **)**

The return statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

6.11. Quit

The quit statement stops execution of a BC program and returns control to UNIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if, for,** or **while** statement.

A Tutorial Introduction to the UNIX Text Editor

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Almost all text input on the UNIX† operating system is done with the text-editor *ed*. This memorandum is a tutorial guide to help beginners get started with text editing.

Although it does not cover everything, it does discuss enough for most users' day-to-day needs. This includes printing, appending, changing, deleting, moving and inserting entire lines of text; reading and writing files; context searching and line addressing; the substitute command; the global commands; and the use of special characters for advanced editing.

September 21, 1978

†UNIX is a Trademark of Bell Laboratories.

A Tutorial Introduction to the UNIX Text Editor

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

Ed is a "text editor", that is, an interactive program for creating and modifying "text", using directions provided by a user at a terminal. The text is often a document like this one, or a program or perhaps data for a program.

This introduction is meant to simplify learning *ed.* The recommended way to learn *ed* is to read this document, simultaneously using *ed* to follow the examples, then to read the description in section I of the *UNIX Programmer's Manual,* all the while experimenting with *ed.* (Solicitation of advice from experienced users is also useful.)

Do the exercises! They cover material not completely discussed in the actual text. An appendix summarizes the commands.

Disclaimer

This is an introduction and a tutorial. For this reason, no attempt is made to cover more than a part of the facilities that *ed* offers (although this fraction includes the most useful and frequently used parts). When you have mastered the Tutorial, try *Advanced Editing on UNIX.* Also, there is not enough space to explain basic UNIX procedures. We will assume that you know how to log on to UNIX, and that you have at least a vague understanding of what a file is. For more on that, read *UNIX for Beginners.*

You must also know what character to type as the end-of-line on your particular terminal. This character is the RETURN key on most terminals. Throughout, we will refer to this character, whatever it is, as RETURN.

Getting Started

We'll assume that you have logged in to your system and it has just printed the prompt character, usually either a $ or a %. The easiest way to get *ed* is to type

```
	ed		(followed by a return)
```

You are now ready to go — *ed* is waiting for you to tell it what to do.

Creating Text — the Append command "a"

As your first problem, suppose you want to create some text starting from scratch. Perhaps you are typing the very first draft of a paper; clearly it will have to start somewhere, and undergo modifications later. This section will show how to get some text in, just to get started. Later we'll talk about how to change it.

When *ed* is first started, it is rather like working with a blank piece of paper — there is no text or information present. This must be supplied by the person using *ed;* it is usually done by typing in the text, or by reading it into *ed* from a file. We will start by typing in some text, and return shortly to how to read files.

First a bit of terminology. In *ed* jargon, the text being worked on is said to be "kept in a buffer." Think of the buffer as a work space, if you like, or simply as the information that you are going to be editing. In effect the buffer is like the piece of paper, on which we will write things, then change some of them, and finally file the whole thing away for another day.

The user tells *ed* what to do to his text by typing instructions called "commands." Most commands consist of a single letter, which must be typed in lower case. Each command is typed on a separate line. (Sometimes the command is preceded by information about what line or lines of text are to be affected — we will discuss these shortly.) *Ed* makes no response to most commands — there is no prompting or typing of messages like "ready". (This silence is preferred by experienced users, but sometimes a hangup for beginners.)

The first command is *append,* written as the letter

```
	a
```

all by itself. It means "append (or add) text lines to the buffer, as I type them in." Appending is rather like writing fresh material on a piece of paper.

So to enter lines of text into the buffer, just type an a followed by a RETURN, followed by the lines of text you want, like this:

```
a
Now is the time
for all good men
to come to the aid of their party.
.
```

The only way to stop appending is to type a line that contains only a period. The "." is used to tell *ed* that you have finished appending. (Even experienced users forget that terminating "." sometimes. If *ed* seems to be ignoring you, type an extra line with just "." on it. You may then find you've added some garbage lines to your text, which you'll have to take out later.)

After the append command has been done, the buffer will contain the three lines

```
Now is the time
for all good men
to come to the aid of their party.
```

The "a" and "." aren't there, because they are not text.

To add more text to what you already have, just issue another a command, and continue typing.

Error Messages — "?"

If at any time you make an error in the commands you type to *ed*, it will tell you by typing

```
?
```

This is about as cryptic as it can be, but with practice, you can usually figure out how you goofed.

Writing text out as a file — the Write command "w"

It's likely that you'll want to save your text for later use. To write out the contents of the buffer onto a file, use the *write* command

```
w
```

followed by the filename you want to write on. This will copy the buffer's contents onto the specified file (destroying any previous information on the file). To save the text on a file named **junk**, for example, type

```
w junk
```

Leave a space between w and the file name. *Ed* will respond by printing the number of characters it wrote out. In this case, *ed* would respond with

```
68
```

(Remember that blanks and the return character at the end of each line are included in the character count.) Writing a file just makes a copy of the text — the buffer's contents are not disturbed, so you can go on adding lines to it. This is an important point. *Ed* at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a w command. (Writing out the text onto a file from time to time as it is being created is a good idea, since if the system crashes or if you make some horrible mistake, you will lose all the text in the buffer but any text that was written onto a file is relatively safe.)

Leaving ed — the Quit command "q"

To terminate a session with *ed*, save the text you're working on by writing it onto a file using the w command, and then type the command

```
q
```

which stands for *quit*. The system will respond with the prompt character ($ or %). At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting.†

Exercise 1:

Enter *ed* and create some text using

```
a
. . . text . . .
.
```

Write it out using w. Then leave *ed* with the q command, and print the file, to see that everything worked. (To print a file, say

```
pr filename
```

or

```
cat filename
```

in response to the prompt character. Try both.)

Reading text from a file — the Edit command "e"

A common way to get text into the buffer is to read it from a file in the file system. This is what you do to edit text that you saved with the w command in a previous session. The *edit* command e fetches the entire contents of a file into the buffer. So if you had saved the three lines "Now is the time", etc., with a w command in an earlier session, the *ed* command

```
e junk
```

would fetch the entire contents of the file **junk** into the buffer, and respond

† Actually, *ed* will print ? if you try to quit without writing. At that point, write if you want; if not, another q will get you out regardless.

```
    68
```

which is the number of characters in **junk**. *If anything was already in the buffer, it is deleted first.*

If you use the e command to read a file into the buffer, then you need not use a file name after a subsequent w command; *ed* remembers the last file name used in an e command, and w will write on this file. Thus a good way to operate is

```
    ed
    e file
    [editing session]
    w
    q
```

This way, you can simply say w from time to time, and be secure in the knowledge that if you got the file name right at the beginning, you are writing into the proper file each time.

You can find out at any time what file name *ed* is remembering by typing the *file* command f. In this example, if you typed

```
    f
```

ed would reply

```
    junk
```

Reading text from a file — the Read command "r"

Sometimes you want to read a file into the buffer without destroying anything that is already there. This is done by the *read* command r. The command

```
    r junk
```

will read the file junk into the buffer; it adds it to the end of whatever is already in the buffer. So if you do a read after an edit:

```
    e junk
    r junk
```

the buffer will contain *two* copies of the text (six lines).

```
    Now is the time
    for all good men
    to come to the aid of their party.
    Now is the time
    for all good men
    to come to the aid of their party.
```

Like the w and e commands, r prints the number of characters read in, after the reading operation is complete.

Generally speaking, r is much less used than e.

Exercise 2:

Experiment with the e command — try reading and printing various files. You may get an error ?name, where name is the name of a file; this means that the file doesn't exist, typically because you spelled the file name wrong, or perhaps that you are not allowed to read or write it. Try alternately reading and appending to see that they work similarly. Verify that

```
    ed filename
```

is exactly equivalent to

```
    ed
    e filename
```

What does

```
    f filename
```

do?

Printing the contents of the buffer — the Print command "p"

To *print* or list the contents of the buffer (or parts of it) on the terminal, use the print command

```
    p
```

The way this is done is as follows. Specify the lines where you want printing to begin and where you want it to end, separated by a comma, and followed by the letter p. Thus to print the first two lines of the buffer, for example, (that is, lines 1 through 2) say

```
    1,2p     (starting line=1, ending line=2 p)
```

Ed will respond with

```
    Now is the time
    for all good men
```

Suppose you want to print *all* the lines in the buffer. You could use 1,3p as above if you knew there were exactly 3 lines in the buffer. But in general, you don't know how many there are, so what do you use for the ending line number? *Ed* provides a shorthand symbol for "line number of last line in buffer" — the dollar sign $. Use it this way:

```
    1,$p
```

This will print *all* the lines in the buffer (line 1 to last line.) If you want to stop the printing before it is finished, push the DEL or Delete key; *ed* will type

```
    ?
```

and wait for the next command.

To print the *last* line of the buffer, you could use

```
$,$p
```

but *ed* lets you abbreviate this to

```
$p
```

You can print any single line by typing the line number followed by a p. Thus

```
1p
```

produces the response

```
Now is the time
```

which is the first line of the buffer.

In fact, *ed* lets you abbreviate even further: you can print any single line by typing *just* the line number — no need to type the letter p. So if you say

```
$
```

ed will print the last line of the buffer.

You can also use $ in combinations like

```
$−1,$p
```

which prints the last two lines of the buffer. This helps when you want to see how far you got in typing.

Exercise 3:

As before, create some text using the a command and experiment with the p command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print a buffer in reverse order by saying

```
3,1p
```

don't work.

The current line — "Dot" or "."

Suppose your buffer still contains the six lines as above, that you have just typed

```
1,3p
```

and *ed* has printed the three lines for you. Try typing just

```
p          (no line numbers)
```

This will print

```
to come to the aid of their party.
```

which is the third line of the buffer. In fact it is the last (most recent) line that you have done anything with. (You just printed it!) You can repeat this p command without line numbers, and it will continue to print line 3.

The reason is that *ed* maintains a record of the last line that you did anything to (in this case, line 3, which you just printed) so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol

```
.          (pronounced "dot").
```

Dot is a line number in the same way that $ is; it means exactly "the current line", or loosely, "the line you most recently did something to." You can use it in several ways — one possibility is to say

```
.,$p
```

This will print all the lines from (including) the current line to the end of the buffer. In our example these are lines 3 through 6.

Some commands change the value of dot, while others do not. The p command sets dot to the number of the last line printed; the last command will set both . and $ to 6.

Dot is most useful when used in combinations like this one:

```
.+1        (or equivalently, .+1p)
```

This means "print the next line" and is a handy way to step slowly through a buffer. You can also say

```
.−1        (or .−1p )
```

which means "print the line *before* the current line." This enables you to go backwards if you wish. Another useful one is something like

```
.−3,.−1p
```

which prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing

```
.=
```

Ed will respond by printing the value of dot.

Let's summarize some things about the p command and dot. Essentially p can be preceded by 0, 1, or 2 line numbers. If there is no line number given, it prints the "current line", the line that dot refers to. If there is one line number given (with or without the letter p), it prints that line (and dot is set there); and if there are two line numbers, it prints all the lines in that range (and sets dot to the last line printed.) If two line numbers are specified the first can't be bigger than the second (see Exercise 2.)

Typing a single return will cause printing of the next line — it's equivalent to .+1p. Try it. Try typing a −; you will find that it's equivalent to .−1p.

Deleting lines: the "d" command

Suppose you want to get rid of the three extra lines in the buffer. This is done by the *delete* command

```
d
```

Except that d deletes lines instead of printing them, its action is similar to that of p. The lines to be deleted are specified for d exactly as they are for p:

starting line, ending line d

Thus the command

```
4,$d
```

deletes lines 4 through the end. There are now three lines left, as you can check by using

```
1,$p
```

And notice that $ now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to $.

Exercise 4:

Experiment with a, e, r, w, p and d until you are sure that you know what they do, and until you understand how dot, $, and line numbers are used.

If you are adventurous, try using line numbers with a, r and w as well. You will find that a will append lines *after* the line number that you specify (rather than after dot); that r reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and that w will write out exactly the lines you specify, not necessarily the whole buffer. These variations are sometimes handy. For instance you can insert a file at the beginning of a buffer by saying

```
0r filename
```

and you can enter lines at the beginning of the buffer by saying

```
0a
. . . text . . .
.
```

Notice that .w is *very* different from

```
.
w
```

Modifying text: the Substitute command "s"

We are now ready to try one of the most important of all commands — the substitute command

```
s
```

This is the command that is used to change individual words or letters within a line or group of lines. It is what you use, for example, for correcting spelling mistakes and typing errors.

Suppose that by a typing error, line 1 says

```
Now is th time
```

— the *e* has been left off *the*. You can use s to fix this up as follows:

```
1s/th/the/
```

This says: "in line 1, substitute for the characters *th* the characters *the*." To verify that it works (*ed* will not print the result automatically) say

```
p
```

and get

```
Now is the time
```

which is what you wanted. Notice that dot must have been set to the line where the substitution took place, since the p command printed that line. Dot is always set this way with the s command.

. The general way to use the substitute command is

starting-line, ending-line s/*change this*/ *to this*/

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between *starting-line* and *ending-line*. Only the first occurrence on each line is changed, however. If you want to change *every* occurrence, see Exercise 5. The rules for line numbers are the same as those for p, except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is *not* changed. This causes an error ? as a warning.)

Thus you can say

```
1,$s/speling/spelling/
```

and correct the first spelling mistake on each line in the text. (This is useful for people who are consistent misspellers!)

If no line numbers are given, the s command assumes we mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence

```
s/something/something else/p
```

which makes some correction on the current line, and then prints it, to make sure it worked out right. If it didn't, you can try again. (Notice that there is a p on the same line as the s command. With few exceptions, p can follow any command; no other multi-command lines are legal.)

It's also legal to say

```
s/ . . . //
```

which means "change the first string of characters to "*nothing*", i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if you had

```
Nowxx is the time
```

you can say

```
s/xx//p
```

to get

```
Now is the time
```

Notice that // (two adjacent slashes) means "no characters", not a blank. There *is* a difference! (See below for another meaning of //.)

Exercise 5:

Experiment with the substitute command. See what happens if you substitute for some word on a line with several occurrences of that word. For example, do this:

```
a
the other side of the coin
.
s/the/on the/p
```

You will get

```
on the other side of the coin
```

A substitute command changes only the first occurrence of the first string. You can change all occurrences by adding a g (for "global") to the s command, like this:

```
s/ . . . / . . . /gp
```

Try other characters instead of slashes to delimit the two sets of characters in the s command — anything should work except blanks or tabs.

(If you get funny results using any of the characters

```
^  .  $  [  *  \  &
```

read the section on "Special Characters".)

Context searching — "/ . . . /"

With the substitute command mastered, you can move on to another highly important idea of *ed* — context searching.

Suppose you have the original three line text in the buffer:

```
Now is the time
for all good men
to come to the aid of their party.
```

Suppose you want to find the line that contains *their* so you can change it to *the.* Now with only three lines in the buffer, it's pretty easy to keep track of what line the word *their* is on. But if the buffer contained several hundred lines, and you'd been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

The way to say "search for a line that contains this particular string of characters" is to type

/string of characters we want to find/

For example, the *ed* command

```
/their/
```

is a context search which is sufficient to find the desired line — it will locate the next occurrence of the characters between slashes ("their"). It also sets dot to that line and prints the line for verification:

```
to come to the aid of their party.
```

"Next occurrence" means that *ed* starts looking for the string at line **.+1**, searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, the search "wraps around" from $ to 1.) It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can't be found in any line, *ed* types the error message

```
?
```

Otherwise it prints the line it found.

You can do both the search for the desired line *and* a substitution all at once, like this:

```
/their/s/their/the/p
```

which will yield

```
to come to the aid of the party.
```

There were three parts to that last command: context search for the desired line, make the substitution, print the line.

The expression **/their/** is a context search expression. In their simplest form, all context search expressions are like this — a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like s. They were used both ways in the examples above.

Suppose the buffer contains the three familiar lines

```
Now is the time
for all good men
to come to the aid of their party.
```

Then the *ed* line numbers

```
/Now/+1
/good/
/party/-1
```

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, you could say

```
/Now/+1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

The choice is dictated only by convenience. You could print all three lines by, for instance

```
/Now/,/party/p
```

or

```
/Now/,/Now/+2p
```

or by any number of similar combinations. The first one of these might be better if you don't know how many lines are involved. (Of course, if there were only three lines in the buffer, you'd use

```
1,$p
```

but not if there were several hundred.)

The basic rule is: a context search expression is *the same as* a line number, so it can be used wherever a line number is needed.

Exercise 6:

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print and delete commands. (They can also be used with r, w, and a.)

Try context searching using ?text? instead of /text/. This scans lines in the buffer in reverse order rather than normal. This is sometimes useful if you go too far while looking for some string of characters — it's an easy way to back up.

(If you get funny results with any of the characters

```
^   .   $   [   *   \   &
```

read the section on "Special Characters".)

Ed provides a shorthand for repeating a context search for the same string. For example, the *ed* line number

```
/string/
```

will find the next occurrence of string. It often happens that this is not the desired line, so the search must be repeated. This can be done by typing merely

```
//
```

This shorthand stands for "the most recently used context search expression." It can also be used as the first string of the substitute command, as in

```
/string1/s//string2/
```

which will find the next occurrence of string1 and replace it by string2. This can save a lot of typing. Similarly

```
??
```

means "scan backwards for the same expression."

Change and Insert — "c" and "i"

This section discusses the *change* command

```
c
```

which is used to change or replace a group of one or more lines, and the *insert* command

```
i
```

which is used for inserting a group of one or more lines.

"Change", written as

```
c
```

is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change lines .+1 through $ to something else, type

```
.+1,$c
. . . type the lines of text you want here . . .
.
```

The lines you type between the c command and the . will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines which have errors in them.

If only one line is specified in the c command, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of . to end the input — this works just like the . in the append command

and must appear by itself on a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

"Insert" is similar to append — for instance

```
/string/i
. . . type the lines to be inserted here . . .
.
```

will insert the given text *before* the next line that contains "string". The text between **i** and **.** is *inserted before* the specified line. If no line number is specified dot is used. Dot is set to the last line inserted.

Exercise 7:

"Change" is rather like a combination of delete followed by insert. Experiment to verify that

```
start, end d
i
. . . text . . .
.
```

is almost the same as

```
start, end c
. . . text . . .
.
```

These are not *precisely* the same if line **$** gets deleted. Check this out. What is dot?

Experiment with **a** and **i**, to see that they are similar, but not the same. You will observe that

```
line-number a
. . . text . . .
.
```

appends *after* the given line, while

```
line-number i
. . . text . . .
.
```

inserts *before* it. Observe that if no line number is given, **i** inserts before line dot, while **a** appends after line dot.

Moving text around: the "m" command

The move command **m** is used for cutting and pasting — it lets you move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer at the end instead. You could do it by saying:

```
1,3w temp
$r temp
1,3d
```

(Do you see why?) but you can do it a lot easier with the **m** command:

```
1,3m$
```

The general case is

```
start line, end line m after this line
```

Notice that there is a third line to be specified — the place where the moved stuff gets put. Of course the lines to be moved can be specified by context searches; if you had

```
First paragraph
. . .
end of first paragraph.
Second paragraph
. . .
end of second paragraph.
```

you could reverse the two paragraphs like this:

```
/Second/,/end of second/m/First/−1
```

Notice the **−1**: the moved text goes *after* the line mentioned. Dot gets set to the last line moved.

The global commands "g" and "v"

The *global* command **g** is used to execute one or more *ed* commands on all those lines in the buffer that match some specified string. For example

```
g/peling/p
```

prints all lines that contain **peling**. More usefully,

```
g/peling/s//pelling/gp
```

makes the substitution everywhere on the line, then prints each corrected line. Compare this to

```
1,$s/peling/pelling/gp
```

which only prints the last line substituted. Another subtle difference is that the **g** command does not give a **?** if **peling** is not found where the **s** command will.

There may be several commands (including **a**, **c**, **i**, **r**, **w**, but not **g**); in that case, every line except the last must end with a backslash \:

```
g/xxx/.−1s/abc/def/\
.+2s/ghi/jkl/\
.−2,.p
```

makes changes in the lines before and after each line that contains **xxx**, then prints all three lines.

The **v** command is the same as **g**, except that the commands are executed on every line that does *not* match the string following **v**:

```
v/ /d
```

deletes every line that does not contain a blank.

Special Characters

You may have noticed that things just don't work right when you used some characters like ., *, $, and others in context searches and the substitute command. The reason is rather complex, although the cure is simple. Basically, *ed* treats these characters as special, with special meanings. For instance, *in a context search or the first string of the substitute command only*, . means "any character," not a period, so

```
/x.y/
```

means "a line with an x, *any character*, and a y," *not* just "a line with an x, a period, and a y." A complete list of the special characters that can cause trouble is the following:

```
^   .   $   [   *   \
```

Warning: The backslash character \ is special to *ed*. For safety's sake, avoid it where possible. If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

```
s/\\\.\*/backslash dot star/
```

will change \.* into "backslash dot star".

Here is a hurried synopsis of the other special characters. First, the circumflex ^ signifies the beginning of a line. Thus

```
/^string/
```

finds string only if it is at the beginning of a line: it will find

```
string
```

but not

```
the string...
```

The dollar-sign $ is just the opposite of the circumflex; it means the end of a line:

```
/string$/
```

will only find an occurrence of string that is at the end of some line. This implies, of course, that

```
/^string$/
```

will find only a line that contains just string, and

```
/^.$/
```

finds a line containing exactly one character.

The character ., as we mentioned above, matches anything;

```
/x.y/
```

matches any of

```
x+y
x−y
x y
x.y
```

This is useful in conjunction with *, which is a repetition character: a* is a shorthand for "any number of a's," so .* matches any number of anythings. This is used like this:

```
s/.*/stuff/
```

which changes an entire line, or

```
s/.*,//
```

which deletes all characters in the line up to and including the last comma. (Since .* finds the longest possible match, this goes up to the last comma.)

[is used with] to form "character classes"; for example,

```
/[0123456789]/
```

matches any single digit — any one of the characters inside the braces will cause a match. This can be abbreviated to [0−9].

Finally, the & is another shorthand character — it is used only on the right-hand part of a substitute command where it means "whatever was matched on the left-hand side". It is used to save typing. Suppose the current line contained

```
Now is the time
```

and you wanted to put parentheses around it. You could just retype the line, but this is tedious. Or you could say

```
s/^/(/
s/$/)/
```

using your knowledge of ^ and $. But the easiest way uses the &:

```
s/.*/(&)/
```

This says "match the whole line, and replace it by itself surrounded by parentheses." The & can be used several times in a line; consider using

```
s/.*/&?  &!!/
```

to produce

```
Now is the time?  Now is the time!!
```

You don't have to match the whole line, of course: if the buffer contains

```
the end of the world
```

you could type

```
/world/s//& is at hand/
```

to produce

```
the end of the world is at hand
```

Observe this expression carefully, for it illustrates how to take advantage of *ed* to save typing. The string **/world/** found the desired line; the shorthand // found the same word in the line; and the & saves you from typing it again.

The & is a special character only within the replacement text of a substitute command, and has no special meaning elsewhere. You can turn off the special meaning of & by preceding it with a \:

```
s/ampersand/\&/
```

will convert the word "ampersand" into the literal symbol & in the current line.

Summary of Commands and Line Numbers

The general form of *ed* commands is the command name, perhaps preceded by one or two line numbers, and, in the case of **e**, **r**, and **w**, followed by a file name. Only one command is allowed per line, but a **p** command may follow any other command (except for **e**, **r**, **w** and **q**).

a: Append, that is, add lines to the buffer (at line dot, unless a different line is specified). Appending continues until . is typed on a new line. Dot is set to the last line appended.

c: Change the specified lines to the new text which follows. The new lines are terminated by a ., as with **a**. If no lines are specified, replace line dot. Dot is set to last line changed.

d: Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless $ is deleted, in which case dot is set to $.

e: Edit new file. Any previous contents of the buffer are thrown away, so issue a **w** beforehand.

f: Print remembered filename. If a name follows **f** the remembered name will be set to it.

g: The command

```
g/---/commands
```

will execute the commands on those lines that contain ---, which can be any context search expression.

i: Insert lines before specified line (or dot) until a . is typed on a new line. Dot is set to last line inserted.

m: Move lines specified to after the line named after **m**. Dot is set to the last line moved.

p: Print specified lines. If none specified, print line dot. A single line number is equivalent to *line-number* **p**. A single return prints .+1, the next line.

q: Quit *ed*. Wipes out all text in buffer if you give it twice in a row without first giving a **w** command.

r: Read a file into buffer (at end unless specified elsewhere.) Dot set to last line read.

s: The command

```
s/string1/string2/
```

substitutes the characters **string1** into **string2** in the specified lines. If no lines are specified, make the substitution in line dot. Dot is set to last line in which a substitution took place, which means that if no substitution took place, dot is not changed. **s** changes only the first occurrence of **string1** on a line; to change all of them, type a **g** after the final slash.

v: The command

```
v/---/commands
```

executes **commands** on those lines that *do not* contain ---.

w: Write out buffer onto a file. Dot is not changed.

.=: Print value of dot. (**=** by itself prints the value of $.)

!: The line

```
!command-line
```

causes **command-line** to be executed as a UNIX command.

/-----/: Context search. Search for next line which contains this string of characters. Print it. Dot is set to the line where string was found. Search starts at .+1, wraps around from $ to 1, and continues to dot, if necessary.

?-----?: Context search in reverse direction. Start search at .−1, scan to 1, wrap around to $.

Advanced Editing on UNIX

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is meant to help secretaries, typists and programmers to mak effective use of the UNIX† facilities for preparing and editing text. It provides explanations and examples of

- special characters, line addressing and global commands in the editor **ed**;
- commands for "cut and paste" operations on files and parts of files, including the **mv**, **cp**, **cat** and **rm** commands, and the **r**, **w**, **m** and **t** commands of the editor;
- editing scripts and editor-based programs like **grep** and **sed**.

Although the treatment is aimed at non-programmers, new users with any background should find helpful hints on how to get their jobs done more easily.

August 4, 1978

†UNIX is a Trademark of Bell Laboratories.

Advanced Editing on UNIX

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

Although UNIX† provides remarkably effective tools for text editing, that by itself is no guarantee that everyone will automatically make the most effective use of them. In particular, people who are not computer specialists — typists, secretaries, casual users — often use the system less effectively than they might.

This document is intended as a sequel to *A Tutorial Introduction to the UNIX Text Editor* [1], providing explanations and examples of how to edit with less effort. (You should also be familiar with the material in *UNIX For Beginners* [2].) Further information on all commands discussed here can be found in *The UNIX Programmer's Manual* [3].

Examples are based on observations of users and the difficulties they encounter. Topics covered include special characters in searches and substitute commands, line addressing, the global commands, and line moving and copying. There are also brief discussions of effective use of related tools, like those for file manipulation, and those based on ed, like grep and sed.

A word of caution. There is only one way to learn to use something, and that is to *use* it. Reading a description is no substitute for trying something. A paper like this one should give you ideas about what to try, but until you actually try something, you will not learn it.

2. SPECIAL CHARACTERS

The editor ed is the primary interface to the system for many people, so it is worthwhile to know how to get the most out of ed for the least effort.

The next few sections will discuss shortcuts and labor-saving devices. Not all of these will be instantly useful to any one person, of course, but a few will be, and the others should give you ideas to store away for future use. And as always, until you try these things, they will remain theoretical knowledge, not something you have confidence in.

†UNIX is a Trademark of Bell Laboratories.

The List command 'l'

ed provides two commands for printing the contents of the lines you're editing. Most people are familiar with p, in combinations like

```
1,$p
```

to print all the lines you're editing, or

```
s/abc/def/p
```

to change 'abc' to 'def' on the current line. Less familiar is the *list* command l (the letter '*l*'), which gives slightly more information than p. In particular, l makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, l will print each tab as ≫ and each backspace as ≪. This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The l command also 'folds' long lines for printing — any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash \, so you can tell it was folded. This is useful for printing long lines on short terminals.

Occasionally the l command will print in a line a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally don't print, like form feed or vertical tab or bell. Each such combination is a single character. When you see such characters, be wary — they may have surprising meanings when printed on some terminals. Often their presence means that your finger slipped while you were typing; you almost never want them.

The Substitute Command 's'

Most of the next few sections will be taken up with a discussion of the substitute command s. Since this is the command for changing the contents of individual lines, it probably has the most complexity of any ed command, and the most potential for effective use.

As the simplest place to begin, recall the meaning of a trailing g after a substitute command. With

```
s/this/that/
```

and

```
s/this/that/g
```

the first one replaces the *first* 'this' on the line with 'that'. If there is more than one 'this' on the line, the second form with the trailing g changes *all* of them.

Either form of the s command can be followed by p or l to 'print' or 'list' (as described in the previous section) the contents of the line:

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

are all legal, and mean slightly different things. Make sure you know what the differences are.

Of course, any s command can be preceded by one or two 'line numbers' to specify that the substitution is to take place on a group of lines. Thus

```
1,$s/mispell/misspell/
```

changes the *first* occurrence of 'mispell' to 'misspell' on every line of the file. But

```
1,$s/mispell/misspell/g
```

changes *every* occurrence in every line (and this is more likely to be what you wanted in this particular case).

You should also notice that if you add a p or l to the end of any of these substitute commands, only the last line that got changed will be printed, not all the lines. We will talk later about how to print all the lines that were modified.

The Undo Command 'u'

Occasionally you will make a substitution in a line, only to realize too late that it was a ghastly mistake. The 'undo' command u lets you 'undo' the last substitution: the last line that was substituted can be restored to its previous state by typing the command

```
u
```

The Metacharacter '.'

As you have undoubtedly noticed when you use ed, certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. In the next several sections, we will talk about these special characters, which are often called 'metacharacters'.

The first one is the period '.'. On the left side of a substitute command, or in a search with '/.../', '.' stands for *any* single character. Thus the search

```
/x.y/
```

finds any line where 'x' and 'y' occur separated by a single character, as in

```
x+y
x-y
x□y
x.y
```

and so on. (We will use □ to stand for a space whenever we need to make it visible.)

Since '.' matches a single character, that gives you a way to deal with funny characters printed by l. Suppose you have a line that, when printed with the l command, appears as

```
....  th\07is  ....
```

and you want to get rid of the \07 (which represents the bell character, by the way).

The most obvious solution is to try

```
s/\07//
```

but this will fail. (Try it.) The brute force solution, which most people would now take, is to re-type the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big, but for a very long line, re-typing is a bore. This is where the metacharacter '.' comes in handy. Since '\07' really represents a single character, if we say

```
s/th.is/this/
```

the job is done. The '.' matches the mysterious character between the 'h' and the 'i', *whatever it is.*

Bear in mind that since '.' matches any single character, the command

```
s/./,/
```

converts the first character on a line into a ',', which very often is not what you intended.

As is true of many characters in ed, the '.' has several meanings, depending on its context. This line shows all three:

```
s/././
```

The first '.' is a line number, the number of the line we are editing, which is called 'line dot'. (We will discuss line dot more in Section 3.) The second '.' is a metacharacter that matches any single character on that line. The third '.' is the only one that really is an honest literal period. On the *right* side of a substitution, '.' is not special. If you apply this command to the line

```
Now is the time.
```

the result will be

```
.ow is the time.
```

which is probably not what you intended.

The Backslash '\'

Since a period means 'any character', the question naturally arises of what to do when you really want a period. For example, how do you convert the line

```
Now is the time.
```

into

```
Now is the time?
```

The backslash '\' does the job. A backslash turns off any special meaning that the next character might have; in particular, '\.' converts the '.' from a 'match anything' into a period, so you can use it to replace the period in

```
Now is the time.
```

like this:

```
s/\./?/
```

The pair of characters '\.' is considered by ed to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains

```
.PP
```

The search

```
/.PP/
```

isn't adequate, for it will find a line like

```
THE APPLICATION OF ...
```

because the '.' matches the letter 'A'. But if you say

```
/\.PP/
```

you will find only lines that contain '.PP'.

The backslash can also be used to turn off special meanings for characters other than '.'. For example, consider finding a line that contains a backslash. The search

```
/\/
```

won't work, because the '\' isn't a literal '\', but instead means that the second '/' no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus

```
/\\/
```

does work. Similarly, you can search for a forward slash '/' with

```
/\//
```

The backslash turns off the meaning of the immediately following '/' so that it doesn't terminate the /.../ construction prematurely.

As an exercise, before reading further, find two substitute commands each of which will convert the line

```
\x\.\y
```

into the line

```
\x\y
```

Here are several solutions; verify that each works as advertised.

```
s/\\\.//
s/x../x/
s/..y/y/
```

A couple of miscellaneous notes about backslashes and special characters. First, you can use any character to delimit the pieces of an s command: there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains a lot of slashes already, like

```
//exec //sys.fort.go // etc...
```

you could use a colon as the delimiter — to delete all the slashes, type

```
s:/::g
```

Second, if # and @ are your character erase and line kill characters, you have to type \# and \@; this is true whether you're talking to ed or any other program.

When you are adding text with a or i or c, backslash is not special, and you should only put in one backslash for each one you really want.

The Dollar Sign '$'

The next metacharacter, the '$', stands for 'the end of the line'. As its most obvious use, suppose you have the line

Now is the

and you wish to add the word 'time' to the end. Use the $ like this:

```
s/$/□time/
```

to get

Now is the time

Notice that a space is needed before 'time' in the substitute command, or you will get

Now is thetime

As another example, replace the second comma in the following line with a period without altering the first:

Now is the time, for all good men,

The command needed is

```
s/,$/./
```

The $ sign here provides context to make specific which comma we mean. Without it, of course, the s command would operate on the first comma to produce

Now is the time. for all good men,

As another example, to convert

Now is the time.

into

Now is the time?

as we did earlier, we can use

```
s/.$/?/
```

Like '.', the '$' has multiple meanings depending on context. In the line

```
$s/$/$/
```

the first '$' refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign, to be added to that line.

The Circumflex '^'

The circumflex (or hat or caret) '^' stands for the beginning of the line. For example, suppose you are looking for a line that begins with 'the'. If you simply say

```
/the/
```

you will in all likelihood find several lines that contain 'the' in the middle before arriving at the one you want. But with

```
/^the/
```

you narrow the context, and thus arrive at the desired one more easily.

The other use of '^' is of course to enable you to insert something at the beginning of a line:

```
s/^/□/
```

places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains *only* the characters

```
.PP
```

you can use the command

```
/^\.PP$/
```

The Star '*'

Suppose you have a line that looks like this:

text x   y *text*

where *text* stands for lots of text, and there are some indeterminate number of spaces between the x and the y. Suppose the job is to replace all the spaces between x and y by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter '*' comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, say

```
s/x□*y/x□y/
```

The construction '□*' means 'as many spaces as possible'. Thus 'x□*y' means 'an x, as many spaces as possible, then a y'.

The star can be used with any character, not just space. If the original example was instead

text x--------y *text*

then all '-' signs can be replaced by a single space with the command

```
s/x-*y/x□y/
```

Finally, suppose that the line was

text x................y *text*

Can you see what trap lies in wait for the unwary? If you blindly type

```
s/x.*y/x□y/
```

what will happen? The answer, naturally, is that it depends. If there are no other x's or y's on the line, then everything works, but it's blind luck, not good management. Remember that '.' matches *any* single character? Then '.*' matches as many single characters as possible, and unless you're careful, it can eat up a lot more of the line than you expected. If the line was, for example, like this:

```
     text x text x................y text y text
```

then saying

```
     s/x.*y/x␣y/
```

will take everything from the *first* 'x' to the *last* 'y', which, in this example, is undoubtedly more than you wanted.

The solution, of course, is to turn off the special meaning of '.' with '\.':

```
     s/x\.*y/x␣y/
```

Now everything works, for '\.*' means 'as many *periods* as possible'.

There are times when the pattern '.*' is exactly what you want. For example, to change

     Now is the time for all good men ....

into

     Now is the time.

use '.*' to eat up everything after the 'for':

```
     s/␣for.*/./
```

There are a couple of additional pitfalls associated with '*' that you should be aware of. Most notable is the fact that 'as many as possible' means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if our line contained

```
     text xy text x          y text
```

and we said

```
     s/x␣*y/x␣y/
```

the *first* 'xy' matches this pattern, for it consists of an 'x', zero spaces, and a 'y'. The result is that the substitute acts on the first 'xy', and does not touch the later one that actually contains some intervening spaces.

The way around this, if it matters, is to specify a pattern like

```
     /x␣␣*y/
```

which says 'an x, a space, then as many more spaces as possible, then a y', in other words, one or more spaces.

The other startling behavior of '*' is again related to the fact that zero is a legitimate number of occurrences of something followed by a star. The command

```
     s/x*/y/g
```

when applied to the line

```
     abcdef
```

produces

```
     yaybycydyeyfy
```

which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there are no x's at the beginning of the line (so that gets converted into a 'y'), nor between the 'a' and the 'b' (so that gets converted into a 'y'), nor ... and so on. Make sure you really want zero matches; if not, in this case write

```
     s/xx*/y/g
```

'xx*' is one or more x's.

The Brackets '[ ]'

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might first think of trying a series of commands like

```
     1,$s/^1*//
     1,$s/^2*//
     1,$s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until finally all numbers are gone, you must get all the digits on one pass. This is the purpose of the brackets [and].

The construction

```
     [0123456789]
```

matches any single digit — the whole thing is called a 'character class'. With a character class, the job is easy. The pattern '[0123456789]*' matches zero or more digits (an entire number), so

```
     1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and just to confuse the issue there are essentially no special characters inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for example, you can say

```
     /[.\$^[]/
```

Within [...], the '[' is not special. To get a ']' into a character class, make it the first character.

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0−9]; similarly, [a−z] stands for the lower case letters, and [A−Z] for upper case.

As a final frill on character classes, you can specify a class that means 'none of the following characters'. This is done by beginning the class with a '^':

```
[^0−9]
```

stands for 'any character *except* a digit'. Thus you might find the first line that doesn't begin with a tab or space by a search like

```
/^[^(space)(tab)]/
```

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that

```
/^[^^]/
```

finds a line that doesn't begin with a circumflex.

The Ampersand '&'

The ampersand '&' is used primarily to save typing. Suppose you have the line

```
Now is the time
```

and you want to make it

```
Now is the best time
```

Of course you can always say

```
s/the/the best/
```

but it seems silly to have to repeat the 'the'. The '&' is used to eliminate the repetition. On the *right* side of a substitute, the ampersand means 'whatever was just matched', so you can say

```
s/the/& best/
```

and the '&' will stand for 'the'. Of course this isn't much of a saving if the thing matched is just 'the', but if it is something truly long or awful, or if it is something like '.*' which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line, regardless of its length,

```
s/.*/(&)/
```

The ampersand can occur more than once on the right side:

```
s/the/& best and & worst/
```

makes

```
Now is the best and the worst time
```

and

```
s/.*/&? &!!/
```

converts the original line into

```
Now is the time? Now is the time!!
```

To get a literal ampersand, naturally the backslash is used to turn off the special meaning:

```
s/ampersand/\&/
```

converts the word into the symbol. Notice that '&' is not special on the left side of a substitute, only on the *right* side.

Substituting Newlines

ed provides a facility for splitting a single line into two or more shorter lines by 'substituting in a newline'. As the simplest example, suppose a line has gotten unmanageably long because of editing (or merely because it was unwisely typed). If it looks like

text   xy   *text*

you can break it between the 'x' and the 'y' like this:

```
s/xy/x\
y/
```

This is actually a single command, although it is typed on two lines. Bearing in mind that '\' turns off special meanings, it seems relatively intuitive that a '\' at the end of a line would make the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word 'very' in a long line by splitting 'very' onto a separate line, and preceding it by the roff or nroff formatting command '.ul'.

text a very big *text*

The command

```
s/□very□/\
.ul\
very\
/
```

converts the line into four shorter lines, preceding the word 'very' by the line '.ul', and eliminating the spaces around the 'very', all at the same time.

When a newline is substituted in, dot is left pointing at the last line created.

Joining Lines

Lines may also be joined together, but this is done with the j command instead of s. Given the lines

```
Now is
□the time
```

and supposing that dot is set to the first of them, then the command

```
j
```

joins them together. No blanks are added, which is why we carefully showed a blank at the beginning of the second line.

All by itself, a j command joins line dot to line dot+1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example,

```
1,$jp
```

joins all the lines into one big one and prints it. (More on line numbers in Section 3.)

Rearranging a Line with \(... \)

(This section should be skipped on first reading.) Recall that '&' is a shorthand that stands for whatever was matched by the left side of an s command. In much the same way you can capture separate pieces of what was matched; the only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose, for instance, that you have a file of lines that consist of names in the form

```
Smith, A. B.
Jones, C.
```

and so on, and you want the initials to precede the name, as in

```
A. B. Smith
C. Jones
```

It is possible to do this with a series of editing commands, but it is tedious and error-prone. (It is instructive to figure out how it is done, though.)

The alternative is to 'tag' the pieces of the pattern (in this case, the last name, and the initials), and then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between \(and \), whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol '\1' refers to whatever matched the first \(...\) pair, '\2' to the second \(...\), and so on.

The command

```
1,$s/^\([^,]*\),␣*\(.*\)/\2␣\1/
```

although hard to read, does the job. The first \(...\) matches the last name, which is any string up to the comma; this is referred to on the right side with '\1'. The second \(...\) is whatever follows the comma and any spaces, and is referred to as '\2'.

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands g and v discussed in section 4 provide a way for you to print exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

3. LINE ADDRESSING IN THE EDITOR

The next general area we will discuss is that of line addressing in ed, that is, how you specify what lines are to be affected by editing commands. We have already used constructions like

```
1,$s/x/y/
```

to specify a change on all lines. And most users are long since familiar with using a single newline (or return) to print the next line, and with

```
/thing/
```

to find a line that contains 'thing'. Less familiar, surprisingly enough, is the use of

```
?thing?
```

to scan *backwards* for the previous occurrence of 'thing'. This is especially handy when you realize that the thing you want to operate on is back up the page from where you are currently editing.

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command.

Address Arithmetic

The next step is to combine the line numbers like '.', '$', '/.../' and '?...?' with '+' and '−'. Thus

```
$−1
```

is a command to print the next to last line of the current file (that is, one line before line '$'). For example, to recall how far you got in a previous editing session,

```
$−5,$p
```

prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six, of course, you'll get an error message.

As another example,

```
.−3,.+3p
```

prints from three lines before where you are now (at line dot) to three lines after, thus giving you a bit of context. By the way, the '+' can be omitted:

```
.−3,.3p
```

is absolutely identical in meaning.

Another area in which you can save typing effort in specifying lines is to use '−' and '+' as line numbers by themselves.

```
−
```

by itself is a command to move back up one line in the file. In fact, you can string several minus signs together to move back up that many lines:

```
−−−
```

moves up three lines, as does '−3'. Thus

```
−3,+3p
```

is also identical to the examples above.

Since '−' is shorter than '.−1', constructions like

```
−,.s/bad/good/
```

are useful. This changes 'bad' to 'good' on the previous line and on the current line.

'+' and '−' can be used in combination with searches using '/.../' and '?...?', and with '$'. The search

```
/thing/−−
```

finds the line containing 'thing', and positions you two lines before it.

Repeated Searches

Suppose you ask for the search

```
/horrible thing/
```

and when the line is printed you discover that it isn't the horrible thing that you wanted, so it is necessary to repeat the search again. You don't have to re-type the search, for the construction

```
//
```

is a shorthand for 'the previous thing that was searched for', whatever it was. This can be repeated as many times as necessary. You can also go backwards:

```
??
```

searches for the same thing, but in the reverse direction.

Not only can you repeat the search, but you can use '//' as the left side of a substitute command, to mean 'the most recent pattern'.

```
/horrible thing/
.... ed prints line with 'horrible thing' ...
s//good/p
```

To go backwards and change a line, say

```
??s//good/
```

Of course, you can still use the '&' on the right hand side of a substitute to stand for whatever got matched:

```
//s//&&/p
```

finds the next occurrence of whatever you searched for last, replaces it by two copies of itself, then prints the line just to verify that it worked.

Default Line Numbers and the Value of Dot

One of the most effective ways to speed up your editing is always to know what lines will be affected by a command if you don't specify the lines it is to act on, and on what line you will be positioned (i.e., the value of dot) when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

As the most obvious example, if you issue a search command like

```
/thing/
```

you are left pointing at the next line that contains 'thing'. Then no address is required with commands like **s** to make a substitution on that line, or **p** to print it, or **l** to list it, or **d** to delete it, or **a** to append text after it, or **c** to change it, or **i** to insert text before it.

What happens if there was no 'thing'? Then you are left right where you were — dot is unchanged. This is also true if you were sitting on the only 'thing' when you issued the command. The same rules hold for searches that use '?...?'; the only difference is the direction in which you search.

The delete command **d** leaves dot pointing at the line that followed the last deleted line. When line '$' gets deleted, however, dot points at the *new* line '$'.

The line-changing commands **a**, **c** and **i** by default all affect the current line — if you give no line number with them, **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

a, **c**, and **i** behave identically in one respect — when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can say

```
a
 ... text ...
 ... botch ...              (minor error)
.
s/botch/correct/            (fix botched line)
a
 ... more text ...
```

without specifying any line number for the substitute command or for the second append command. Or you can say

```
a
... text ...
... horrible botch ...          (major error)
.
c                               (replace entire line)
... fixed up line ...
```

You should experiment to determine what happens if you add *no* lines with a, c or i.

The r command will read a file into the text being edited, either at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even say 0r to read a file in at the beginning of the text. (You can also say 0a or 1i to start adding text at the beginning.)

The w command writes out the entire file. If you precede the command by one line number, that line is written, while if you precede it by two line numbers, that range of lines is written. The w command does *not* change dot: the current line remains the same, regardless of what lines are written. This is true even if you say something like

```
/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

Since the w command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you do something foolish, like clobbering what you're editing.

The least intuitive behavior, in a sense, is that of the s command. The rule is simple — you are left sitting on the last line that got changed. If there were no changes, then dot is unchanged.

To illustrate, suppose that there are three lines in the buffer, and you are sitting on the middle one:

```
x1
x2
x3
```

Then the command

```
−,+s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, then the result would be to change and print only the first line, and that is where dot would be set.

Semicolon ';'

Searches with '/.../' and '?...?' start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like this:

```
.
.
.
ab
.
.
.
bc
.
.
```

Starting at line 1, one would expect that the command

```
/a/,/b/p
```

prints all the lines from the 'ab' to the 'bc' inclusive. Actually this is not what happens. *Both* searches (for 'a' and for 'b') start from the same point, and thus they both find the line that contains 'ab'. The result is to print a single line. Worse, if there had been a line with a 'b' in it before the 'ab' line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order.

This is because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In ed, the semicolon ';' can be used just like comma, with the single difference that use of a semicolon forces dot to be set at that point as the line numbers are being evaluated. In effect, the semicolon 'moves' dot. Thus in our example above, the command

```
/a/;/b/p
```

prints the range of lines from 'ab' to 'bc', because after the 'a' is found, dot is set to that line, and then 'b' is searched for, starting beyond that line.

This property is most often useful in a very simple situation. Suppose you want to find the *second* occurrence of 'thing'. You could say

```
/thing/
//
```

but this prints the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to say

```
/thing/;//
```

This says to find the first occurrence of 'thing', set dot to that line, then find the second and print only that.

Closely related is searching for the second previous occurrence of something, as in

```
?something?;??
```

Printing the third or fourth or ... in either direction is left as an exercise.

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to say

```
1;/thing/
```

because this fails if 'thing' occurs on line 1. But it is possible to say

```
0;/thing/
```

(one of the few places where 0 is a legal line number), for this starts the search at line 1.

Interrupting the Editor

As a final note on what dot gets set to, you should be aware that if you hit the interrupt or delete or rubout or break key while ed is doing a command, things are put back together again and your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable — if you are reading or writing a file or making substitutions or deleting lines, these will be stopped in some clean but unpredictable state in the middle (which is why it is not usually wise to stop them). Dot may or may not be changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus if you print until you see an interesting line, then hit delete, you are *not* sitting on that line or even near it. Dot is left where it was when the p command was started.

4. GLOBAL COMMANDS

The global commands g and v are used to perform one or more editing commands on all lines that either contain (g) or don't contain (v) a specified pattern.

As the simplest example, the command

```
g/UNIX/p
```

prints all lines that contain the word 'UNIX'. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command: exactly the same rules and limitations apply.

As another example, then,

```
g/^\./p
```

prints all the formatting commands in a file (lines that begin with '.').

The v command is identical to g, except that it operates on those line that do *not* contain an occurrence of the pattern. (Don't look too hard for mnemonic significance to the letter 'v'.) So

```
v/^\./p
```

prints all the lines that don't begin with '.' — the actual text lines.

The command that follows g or v can be anything:

```
g/^\./d
```

deletes all lines that begin with '.', and

```
g/^$/d
```

deletes all empty lines.

Probably the most useful command that can follow a global is the substitute command, for this can be used to make a change and print each affected line for verification. For example, we could change the word 'Unix' to 'UNIX' everywhere, and verify that it really worked, with

```
g/Unix/s//UNIX/gp
```

Notice that we used '//' in the substitute command to mean 'the previous pattern', in this case, 'Unix'. The p command is done on every line that matches the pattern, not just those on which a substitution took place.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line in turn is examined, dot is set to that line, and the command executed. This means that it is possible for the command that follows a g or v to use addresses, set dot, and so on, quite freely.

```
g/^\.PP/+
```

prints the line that follows each '.PP' command (the signal for a new paragraph in some formatting packages). Remember that '+' means 'one line past dot'. And

```
g/topic/?^\.SH?1
```

searches for each line that contains 'topic', scans backwards until it finds a line that begins '.SH' (a section heading) and prints the line that follows that, thus showing the section headings under which 'topic' is mentioned. Finally,

```
g/^\.EQ/+,/^\.EN/-p
```

prints all the lines that lie between lines beginning with '.EQ' and '.EN' formatting commands.

The g and v commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

Multi-line Global Commands

It is possible to do more than one command under the control of a global command, although the syntax for expressing the operation is not especially natural or pleasant. As an example, suppose the task is to change 'x' to 'y' and 'a' to 'b' on all lines that contain 'thing'. Then

```
g/thing/s/x/y/\
s/a/b/
```

is sufficient. The '\' signals the g command that the set of commands continues on the next line; it terminates on the first line that does not end with '\'. (As a minor blemish, you can't use a substitute command to insert a newline within a g command.)

You should watch out for this problem: the command

```
g/x/s//y/\
s/a/b/
```

does *not* work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be 'x' (as expected), and sometimes it will be 'a' (not expected). You must spell it out, like this:

```
g/x/s/x/y/\
s/a/b/
```

It is also possible to execute a, c and i commands under a global command; as with other multi-line constructions, all that is needed is to add a '\' at the end of each line except the last. Thus to add a '.nf' and '.sp' command before each '.EQ' line, type

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a '.' to terminate the i command, unless there are further commands being done under the global. On the other hand, it does no harm to put it in either.

5. CUT AND PASTE WITH UNIX COMMANDS

One editing area in which non-programmers seem not very confident is in what might be called 'cut and paste' operations — changing the name of a file, making a copy of a file somewhere else, moving a few lines from one place to another in a file, inserting one file in the middle of another, splitting a file into pieces, and splicing two or more files together.

Yet most of these operations are actually quite easy, if you keep your wits about you and go cautiously. The next several sections talk about cut and paste. We will begin with the UNIX commands for moving entire files around, then discuss **ed** commands for operating on pieces of files.

Changing the Name of a File

You have a file named 'memo' and you want it to be called 'paper' instead. How is it done?

The UNIX program that renames files is called **mv** (for 'move'); it 'moves' the file from one name to another, like this:

```
mv memo paper
```

That's all there is to it: **mv** from the old name to the new name.

```
mv oldname newname
```

Warning: if there is already a file around with the new name, its present contents will be silently clobbered by the information from the other file. The one exception is that you can't move a file to itself —

```
mv x x
```

is illegal.

Making a Copy of a File

Sometimes what you want is a copy of a file — an entirely fresh version. This might be because you want to work on a file, and yet save a copy in case something gets fouled up, or just because you're paranoid.

In any case, the way to do it is with the **cp** command. (**cp** stands for 'copy'; the system is big on short command names, which are appreciated by heavy users, but sometimes a strain for novices.) Suppose you have a file called 'good' and you want to save a copy before you make some dramatic editing changes. Choose a name — 'savegood' might be acceptable — then type

```
cp good savegood
```

This copies 'good' onto 'savegood', and you now have two identical copies of the file 'good'. (If 'savegood' previously contained something, it gets overwritten.)

Now if you decide at some time that you want to get back to the original state of 'good', you can say

```
mv savegood good
```

(if you're not interested in 'savegood' any more), or

```
cp savegood good
```

if you still want to retain a safe copy.

In summary, **mv** just renames a file; **cp** makes a duplicate copy. Both of them clobber the 'target' file if it already exists, so you had better be sure that's what you want to do *before* you do it.

Removing a File

If you decide you are really done with a file forever, you can remove it with the **rm** command:

```
rm savegood
```

throws away (irrevocably) the file called 'savegood'.

Putting Two or More Files Together

The next step is the familiar one of collecting two or more files into one big one. This will be needed, for example, when the author of a paper decides that several sections need to be combined into one. There are several ways to do it, of which the cleanest, once you get used to it, is a program called cat. (Not *all* programs have two-letter names.) **cat** is short for 'concatenate', which is exactly what we want to do.

Suppose the job is to combine the files 'file1' and 'file2' into a single file called 'bigfile'. If you say

```
cat file
```

the contents of 'file' will get printed on your terminal. If you say

```
cat file1 file2
```

the contents of 'file1' and then the contents of 'file2' will *both* be printed on your terminal, in that order. So **cat** combines the files, all right, but it's not much help to print them on the terminal — we want them in 'bigfile'.

Fortunately, there is a way. You can tell the system that instead of printing on your terminal, you want the same information put in a file. The way to do it is to add to the command line the character > and the name of the file where you want the output to go. Then you can say

```
cat file1 file2 >bigfile
```

and the job is done. (As with **cp** and **mv**, you're putting something into 'bigfile', and anything that was already there is destroyed.)

This ability to 'capture' the output of a program is one of the most useful aspects of the system. Fortunately it's not limited to the **cat** program — you can use it with *any* program that prints on your terminal. We'll see some more uses for it in a moment.

Naturally, you can combine several files, not just two:

```
cat file1 file2 file3 ... >bigfile
```

collects a whole bunch.

Question: is there any difference between

```
cp good savegood
```

and

```
cat good >savegood
```

Answer: for most purposes, no. You might reasonably ask why there are two programs in that case, since **cat** is obviously all you need. The answer is that **cp** will do some other things as well, which you can investigate for yourself by reading the manual. For now we'll stick to simple usages.

Adding Something to the End of a File

Sometimes you want to add one file to the end of another. We have enough building blocks now that you can do it; in fact before reading further it would be valuable if you figured out how. To be specific, how would you use **cp**, **mv** and/or **cat** to add the file 'good1' to the end of the file 'good'?

You could try

```
cat good good1 >temp
mv temp good
```

which is probably most direct. You should also understand why

```
cat good good1 >good
```

doesn't work. (Don't practice with a good 'good'!)

The easy way is to use a variant of >, called >>. In fact, >> is identical to > except that instead of clobbering the old file, it simply tacks stuff on at the end. Thus you could say

```
cat good1 >>good
```

and 'good1' is added to the end of 'good'. (And if 'good' didn't exist, this makes a copy of 'good1' called 'good'.)

6. CUT AND PASTE WITH THE EDITOR

Now we move on to manipulating pieces of files — individual lines or groups of lines. This is another area where new users seem unsure of themselves.

Filenames

The first step is to ensure that you know the ed commands for reading and writing files. Of course you can't go very far without knowing r and w. Equally useful, but less well known, is the 'edit' command e. Within ed, the command

```
e newfile
```

says 'I want to edit a new file called *newfile*, without leaving the editor.' The e command discards whatever you're currently working on and starts over on *newfile*. It's exactly the same as if you had quit with the q command, then re-entered ed with a new file name, except that if you have a pattern remembered, then a command like // will still work.

If you enter ed with the command

```
ed file
```

ed remembers the name of the file, and any subsequent e, r or w commands that don't contain a filename will refer to this remembered file. Thus

```
ed file1
 ... (editing) ...
w         (writes back in file1)
e file2   (edit new file, without leaving editor)
 ... (editing on file2) ...
w         (writes back on file2)
```

(and so on) does a series of edits on various files without ever leaving ed and without typing the name of any file more than once. (As an aside, if you examine the sequence of commands here, you can see why many UNIX systems use e as a synonym for ed.)

You can find out the remembered file name at any time with the f command; just type f without a file name. You can also change the name of the remembered file name with f; a useful sequence is

```
ed precious
f junk
... (editing) ...
```

which gets a copy of a precious file, then uses f to guarantee that a careless w command won't clobber the original.

Inserting One File into Another

Suppose you have a file called 'memo', and you want the file called 'table' to be inserted just after the reference to Table 1. That is, in 'memo' somewhere is a line that says

```
Table 1 shows that ...
```

and the data contained in 'table' has to go there, probably so it will be formatted properly by nroff or troff. Now what?

This one is easy. Edit 'memo', find 'Table 1', and add the file 'table' right there:

```
ed memo
/Table 1/
Table 1 shows that ... [response from ed]
.r table
```

The critical line is the last one. As we said earlier, the r command reads a file; here you asked for it to be read in right after line dot. An r command without any address adds lines at the end, so it is the same as $r.

Writing out Part of a File

The other side of the coin is writing out part of the document you're editing. For example, maybe you want to split out into a separate file that table from the previous example, so it can be formatted and tested separately. Suppose that in the file being edited we have

```
.TS
...[lots of stuff]
.TE
```

which is the way a table is set up for the tbl program. To isolate the table in a separate file called 'table', first find the start of the table (the '.TS' line), then write out the interesting part:

```
/^\.TS/
.TS [ed prints the line it found]
.,/^\.TE/w table
```

and the job is done. If you are confident, you can do it all at once with

```
/^\.TS/;/^\.TE/w table
```

The point is that the w command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. For example, if you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed later, then save it — don't re-type it. In the editor, say

```
a
...lots of stuff...
...horrible line...
.
.w  temp
a
...more stuff...
.
.r temp
a
...more stuff...
.
```

This last example is worth studying, to be sure you appreciate what's going on.

Moving Lines Around

Suppose you want to move a paragraph from its present position in a paper to the end. How would you do it? As a concrete example, suppose each paragraph in the paper begins with the formatting command '.PP'. Think about it and write down the details before reading on.

The brute force way (not necessarily bad) is to write the paragraph onto a temporary file, delete it from its current position, then read in the temporary file at the end. Assuming that you are sitting on the '.PP' command that begins the paragraph, this is the sequence of commands:

```
.,/^\.PP/−w temp
.,//−d
$r temp
```

That is, from where you are now ('.') until one line before the next '.PP' ('/^\.PP/−') write onto 'temp'. Then delete the same lines. Finally, read 'temp' at the end.

As we said, that's the brute force way. The easier way (often) is to use the *move* command **m** that **ed** provides — it lets you do the whole set of operations at one crack, without any temporary file.

The **m** command is like many other **ed** commands in that it takes up to two line numbers in front that tell what lines are to be affected. It is also *followed* by a line number that tells where the lines are to go. Thus

```
line1, line2 m line3
```

says to move all the lines between 'line1' and 'line2' after 'line3'. Naturally, any of 'line1' etc., can be patterns between slashes, $ signs, or other ways to specify lines.

Suppose again that you're sitting at the first line of the paragraph. Then you can say

```
.,/^\.PP/−m$
```

That's all.

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first one to after the second. Suppose that you are positioned at the first. Then

```
m+
```

does it. It says to move line dot to after one line after line dot. If you are positioned on the second line,

```
m−−
```

does the interchange.

As you can see, the **m** command is more succinct and direct than writing, deleting and re-reading. When is brute force better anyway? This is a matter of personal taste — do what you have most confidence in. The main difficulty with the **m** command is that if you use patterns to specify both the lines you are moving and the target, you have to take care that you specify them properly, or you may well not move the lines you thought you did. The result of a botched **m** command can be a ghastly mess. Doing the job a step at a time makes it easier for you to verify at each step that you accomplished what you wanted to. It's also a good idea to issue a **w** command before doing anything complicated; then if you goof, it's easy to back up to where you were.

Marks

ed provides a facility for marking a line with a particular name so you can later reference it by name regardless of its actual line number. This can be handy for moving lines, and for keeping track of them as they move. The *mark* command is **k**; the command

```
kx
```

marks the current line with the name 'x'. If a line number precedes the **k**, that line is marked. (The mark name must be a single lower case letter.) Now you can refer to the marked line with the address

```
'x
```

Marks are most useful for moving things around. Find the first line of the block to be moved, and mark it with *'a*. Then find the last line and mark it with *'b*. Now position yourself at the place where the stuff is to go and say

```
'a,'bm.
```

Bear in mind that only one line can have a particular mark name associated with it at any given time.

Copying Lines

We mentioned earlier the idea of saving a line that was hard to type or used often, so as to cut down on typing time. Of course this could be more than one line; then the saving is presumably even greater.

ed provides another command, called t (for 'transfer') for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The t command is identical to the m command, except that instead of moving lines it simply duplicates them at the place you named. Thus

```
1,$t$
```

duplicates the entire contents that you are editing. A more common use for t is for creating a series of lines that differ only slightly. For example, you can say

```
a
.......... x ......... (long line)
.
t.             (make a copy)
s/x/y/         (change it a bit)
t.             (make third copy)
s/y/z/         (change it a bit)
```

and so on.

The Temporary Escape '!'

Sometimes it is convenient to be able to temporarily escape from the editor to do some other UNIX command, perhaps one of the file copy or move commands discussed in section 5, without leaving the editor. The 'escape' command ! provides a way to do this.

If you say

```
!any UNIX command
```

your current editing state is suspended, and the UNIX command you asked for is executed. When the command finishes, ed will signal you by printing another !; at that point you can resume editing.

You can really do *any* UNIX command, including another ed. (This is quite common, in fact.) In this case, you can even do another !.

7. SUPPORTING TOOLS

There are several tools and techniques that go along with the editor, all of which are relatively easy once you know how ed works, because they are all based on the editor. In this section we will give some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. More information on each can be found in [3].

Grep

Sometimes you want to find all occurrences of some word or pattern in a set of files, to edit them or perhaps just to verify their presence or absence. It may be possible to edit each file separately and look for the pattern of interest, but if there are many files this can get very tedious, and if the files are really big, it may be impossible because of limits in ed.

The program grep was invented to get around these limitations. The search patterns that we have described in the paper are often called 'regular expressions', and 'grep' stands for

```
g/re/p
```

That describes exactly what grep does — it prints every line in a set of files that contains a particular pattern. Thus

```
grep 'thing' file1 file2 file3 ...
```

finds 'thing' wherever it occurs in any of the files 'file1', 'file2', etc. grep also indicates the file in which the line was found, so you can later edit it if you like.

The pattern represented by 'thing' can be any pattern you can use in the editor, since grep and ed use exactly the same mechanism for pattern searching. It is wisest always to enclose the pattern in the single quotes '...' if it contains any non-alphabetic characters, since many such characters also mean something special to the UNIX command interpreter (the 'shell'). If you don't quote them, the command interpreter will try to interpret them before grep gets a chance.

There is also a way to find lines that *don't* contain a pattern:

```
grep −v 'thing' file1 file2 ...
```

finds all lines that don't contains 'thing'. The −v must occur in the position shown. Given grep and grep −v, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain 'x' but not 'y':

```
grep x file... | grep −v y
```

(The notation | is a 'pipe', which causes the output of the first command to be used as input to the second command: see [2].)

Editing Scripts

If a fairly complicated set of editing operations is to be done on a whole set of files, the easiest thing to do is to make up a 'script', i.e., a file that contains the operations you want to perform, then apply this script to each file in turn.

For example, suppose you want to change every 'Unix' to 'UNIX' and every 'Gcos' to 'GCOS' in a large number of files. Then put into the file 'script' the lines

```
g/Unix/s//UNIX/g
g/Gcos/s//GCOS/g
w
q
```

Now you can say

```
ed file1 <script
ed file2 <script
...
```

This causes ed to take its commands from the prepared script. Notice that the whole job has to be planned in advance.

And of course by using the UNIX command interpreter, you can cycle through a set of files automatically, with varying degrees of ease.

Sed

sed ('stream editor') is a version of the editor with restricted capabilities but which is capable of processing unlimited amounts of input. Basically sed copies its input to its output, applying one or more editing commands to each line of input.

As an example, suppose that we want to do the 'Unix' to 'UNIX' part of the example given above, but without rewriting the files. Then the command

```
sed 's/Unix/UNIX/g' file1 file2 ...
```

applies the command 's/Unix/UNIX/g' to all lines from 'file1', 'file2', etc., and copies all lines to the output. The advantage of using sed in such a case is that it can be used with input too large for ed to handle. All the output can be collected in one place, either in a file or perhaps piped into another program.

If the editing transformation is so complicated that more than one editing command is needed, commands can be supplied from a file, or on the command line, with a slightly more complex syntax. To take commands from a file, for example,

```
sed −f cmdfile input−files...
```

sed has further capabilities, including conditional testing and branching, which we cannot go into here.

Acknowledgement

I am grateful to Ted Dolotta for his careful reading and valuable suggestions.

References

[1] Brian W. Kernighan, *A Tutorial Introduction to the UNIX Text Editor.* Bell Laboratories internal memorandum.

[2] Brian W. Kernighan, *UNIX For Beginners.* Bell Laboratories internal memorandum.

[3] Ken L. Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual.* Bell Laboratories.

Edit: A Tutorial

Ricki Blau

James Joyce

Computing Services
University of California
Berkeley, California 94720

ABSTRACT

This narrative introduction to the use of the text editor *edit* assumes no prior familiarity with computers or with text editing. Its aim is to lead the beginning UNIX† user through the fundamental steps of writing and revising a file of text. Edit, a version of the text editor *ex,* was designed to provide an informative environment for new and casual users.

We welcome comments and suggestions about this tutorial and the UNIX documentation in general.

September 1981

†UNIX is a trademark of Bell Laboratories.

Contents

Introduction

Text editing using a terminal connected to a computer allows you to create, modify, and print text easily. A *text editor* is a program that assists you as you create and modify text. The text editor you will learn here is named *edit.* Creating text using edit is as easy as typing it on an electric typewriter. Modifying text involves telling the text editor what you want to add, change, or delete. You can review your text by typing a command to print the file contents as they were entered by you. Another program, a text formatter, rearranges your text for you into "finished form." This document does not discuss the use of a text formatter.

These lessons assume no prior familiarity with computers or with text editing. They consist of a series of text editing sessions which lead you through the fundamental steps of creating and revising text. After scanning each lesson and before beginning the next, you should practice the examples at a terminal to get a feeling for the actual process of text editing. If you set aside some time for experimentation, you will soon become familiar with using the computer to write and modify text. In addition to the actual use of the text editor, other features of UNIX will be very important to your work. You can begin to learn about these other features by reading "Communicating with UNIX" or one of the other tutorials that provide a general introduction to the system. You will be ready to proceed with this lesson as soon as you are familiar with (1) your terminal and its special keys, (2) the login procedure, (3) and the ways of correcting typing errors. Let's first define some terms:

program	A set of instructions, given to the computer, describing the sequence of steps the computer performs in order to accomplish a specific task. The tasks must be specific, such as balancing your checkbook or editing your text. A general task, such as working for world peace, is something we can do, but not something we can write programs to do.
UNIX	UNIX is a special type of program, called an operating system, that supervises the machinery and all other programs comprising the total computer system.
edit	*edit* is the name of the UNIX text editor you will be learning to use, and is a program that aids you in writing or revising text. Edit was designed for beginning users, and is a simplified version of an editor named *ex.*
file	Each UNIX account is allotted space for the permanent storage of information, such as programs, data or text. A file is a logical unit of data, for example, an essay, a program, or a chapter from a book, which is stored on a computer system. Once you create a file, it is kept until you instruct the system to remove it. You may create a file during one UNIX session, end the session, and return to use it at a later time. Files contain anything you choose to write and store in them. The sizes of files vary to suit your needs; one file might hold only a single number, yet another might contain a very long document or program. The only way to save information from one session to the next is to store it in a file, which you will learn in Session 1.
filename	Filenames are used to distinguish one file from another, serving the same purpose as the labels of manila folders in a file cabinet. In order to write or access information in a file, you use the name of that file in a UNIX command, and the system will automatically locate the file.
disk	Files are stored on an input/output device called a disk, which looks something like a stack of phonograph records. Each surface is coated with a material similar to the coating on magnetic recording tape, and information is recorded on it.
buffer	A temporary work space, made available to the user for the duration of a session of text editing and used for creating and modifying the text file. We can think of the buffer as a blackboard that is erased after each class, where each session with the editor is a class.

Session 1

Making contact with UNIX

To use the editor you must first make contact with the computer by logging in to UNIX. We'll quickly review the standard UNIX login procedure for the two ways you can make contact: on a terminal that is directly linked to the computer, or over a telephone line where the computer answers your call.

Directly-linked terminals

Turn on your terminal and press the RETURN key. You are now ready to login.

Dial-up terminals

If your terminal connects with the computer over a telephone line, turn on the terminal, dial the system access number, and, when you hear a high-pitched tone, place the receiver of the telephone in the acoustic coupler. You are now ready to login.

Logging in

The message inviting you to login is:

```
        :login:
```

Type your login name, which identifies you to UNIX, on the same line as the login message, and press RETURN. If the terminal you are using has both upper and lower case, **be sure you enter your login name in lower case;** otherwise UNIX assumes your terminal has only upper case and will not recognize lower case letters you may type. UNIX types ":login:" and you reply with your login name, for example "susan":

:login: **susan** *(and press the RETURN key)*

(In the examples, input you would type appears in **bold face** to distinguish it from the responses from UNIX.)

UNIX will next respond with a request for a password as an additional precaution to prevent unauthorized people from using your account. The password will not appear when you type it, to prevent others from seeing it. The message is:

Password: *(type your password and press RETURN)*

If any of the information you gave during the login sequence was mistyped or incorrect, UNIX will respond with

```
        Login incorrect.
        :login:
```

in which case you should start the login process anew. Assuming that you have successfully logged in, UNIX will print the message of the day and eventually will present you with a % at the beginning of a fresh line. The % is the UNIX prompt symbol which tells you that UNIX is ready to accept a command.

Asking for *edit*

You are ready to tell UNIX that you want to work with edit, the text editor. Now is a convenient time to choose a name for the file of text you are about to create. To begin your editing session, type **edit** followed by a space and then the filename you have selected; for example, "text". When you have completed the command, press the RETURN key and wait for edit's response:

```
% **edit text**     *(followed by a RETURN)*
"text" No such file or directory
:
```

If you typed the command correctly, you will now be in communication with edit. Edit has set aside a buffer for use as a temporary working space during your current editing session. It also checked to see if the file you named, "text", already existed. It was unable to find such a file, since "text" is a new file we are about to create. Edit confirms this with the line:

```
"text" No such file or directory
```

On the next line appears edit's prompt ":", announcing that you are in *command mode* and edit expects a command from you. You may now begin to create the new file.

The "Command not found" message

If you misspelled edit by typing, say, "editor", your request would be handled as follows:

```
% editor
editor: Command not found
%
```

Your mistake in calling edit "editor" was treated by UNIX as a request for a program named "editor". Since there is no program named "editor", UNIX reported that the program was "not found". A new % indicates that UNIX is ready for another command, and you may then enter the correct command.

A summary

Your exchange with UNIX as you logged in and made contact with edit should look something like this:

```
:login: susan
Password:
... A Message of General Interest ...
% edit text
"text" No such file or directory
:
```

Entering text

You may now begin entering text into the buffer. This is done by *appending* (or adding) text to whatever is currently in the buffer. Since there is nothing in the buffer at the moment, you are appending text to nothing; in effect, since you are adding text to nothing you are creating text. Most edit commands have two forms: a word that suggests what the command does, and a shorter abbreviation of that word. Either form may be used. Many beginners find the full command names easier to remember at first, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is "append", and it may be abbreviated "a". Type **append** and press the RETURN key.

```
% edit text
:append
```

Messages from *edit*

If you make a mistake in entering a command and type something that edit does not recognize, edit will respond with a message intended to help you diagnose your error. For example, if you misspell the command to input text by typing, perhaps, "add" instead of "append" or "a", you will receive this message:

```
:add
add: Not an editor command
:
```

When you receive a diagnostic message, check what you typed in order to determine what part of your command confused edit. The message above means that edit was unable to recognize your mistyped command and, therefore, did not execute it. Instead, a new ":" appeared to let you know that edit is again ready to execute a command.

Text input mode

By giving the command "append" (or using the abbreviation "a"), you entered *text input mode,* also known as *append mode.* When you enter text input mode, edit stops sending you a prompt. You will not receive any prompts or error messages while in text input mode. You can enter pretty much anything you want on the lines. The lines are transmitted one by one to the buffer and held there during the editing session. You may append as much text as you want, and *when you wish to stop entering text lines you should type a period as the only character on the line and press the RETURN key.* When you type the period and press RETURN, you signal that you want to stop appending text, and edit responds by allowing you to exit text input mode and reenter command mode. Edit will again prompt you for a command by printing ":".

Leaving append mode does not destroy the text in the buffer. You have to leave append mode to do any of the other kinds of editing, such as changing, adding, or printing text. If you type a period as the first character and type any other character on the same line, edit will believe you want to remain in append mode and will not let you out. As this can be very frustrating, be sure to type **only** the period and the RETURN key.

This is a good place to learn an important lesson about computers and text: a blank space is a character as far as a computer is concerned. If you so much as type a period followed by a blank (that is, type a period and then the space bar on the keyboard), you will remain in append mode with the last line of text being:

```
. 
```

Let's say that the lines of text you enter are (try to type **exactly** what you see, including "thiss"):

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
.
```

The last line is the period followed by a RETURN that gets you out of append mode.

Making corrections

If you have read a general introduction to UNIX, such as "Communicating with UNIX", you will recall that it is possible to erase individual letters that you have typed. This is done by typing the designated erase character as many times as there are characters you want to erase.

The usual erase character is the backspace (control-H), and you can correct typing errors in the line you are typing by holding down the CTRL key and typing the "H" key. If you try typing control-H you will notice that the terminal backspaces in the line you are on. You can backspace over your error, and then type what you want to be the rest of the line.

If you make a bad start in a line and would like to begin again, you can either backspace to the beginning of the line or you can use the at-sign "@" to erase everything on the line:

```
Text edtiing is strange, but@
Text editing is strange, but nice.
```

When you type the at-sign (@), you erase the entire line typed so far and are given a fresh line to type on. You may immediately begin to retype the line. This, unfortunately, does not help after you type the line and press RETURN. To make corrections in lines that have been completed, it is necessary to use the editing commands covered in the next session and those that follow.

Writing text to disk

You are now ready to edit the text. The simplest kind of editing is to write it to disk as a file for safekeeping after the session is over. This is the only way to save information from one session to the next, since the editor's buffer is temporary and will last only until the end of the editing session. Learning how to write a file to disk is second in importance only to entering the text. To write the contents of the buffer to a disk file, use the command "write" (or its abbreviation "w"):

```
:**write**
```

Edit will copy the contents of the buffer to a disk file. If the file does not yet exist, a new file will be created automatically and the presence of a "[New file]" will be noted. The newly-created file will be given the name specified when you entered the editor, in this case "text". To confirm that the disk file has been successfully written, edit will repeat the filename and give the number of lines and the total number of characters in the file. The buffer remains unchanged by the "write" command. All of the lines that were written to disk will still be in the buffer, should you want to modify or add to them.

Edit must have a filename to use before it can write a file. If you forgot to indicate the name of the file when you began the editing session, edit will print

```
No current filename
```

in response to your write command. If this happens, you can specify the filename in a new write command:

```
:write text
```

After the "write" (or "w"), type a space and then the name of the file.

Signing off

We have done enough for this first lesson on using the UNIX text editor, and are ready to quit the session with edit. To do this we type "quit" (or "q") and press RETURN:

```
:write
"text" [New file]  3 lines, 90 characters
:quit
%
```

The % is from UNIX to tell you that your session with edit is over and you may command UNIX further. Since we want to end the entire session at the terminal, we also need to exit from UNIX. In response to the UNIX prompt of "%" type the command

```
%logout
```

This will end your session with UNIX, and will ready the terminal for the next user. It is always important to type **logout** at the end of a session to make absolutely sure no one could accidentally stumble into your abandoned session and thus gain access to your files, tempting even the most honest of souls.

This is the end of the first session on UNIX text editing.

Session 2

Login with UNIX as in the first session:

```
:login: susan  (carriage return)
Password:      (give password and carriage return)
... A Message of General Interest ...
%
```

When you indicate you want to edit, you can specify the name of the file you worked on last time. This will start edit working, and it will fetch the contents of the file into the buffer, so that you can resume editing the same file. When edit has copied the file into the buffer, it will repeat its name and tell you the number of lines and characters it contains. Thus,

```
% edit text
"text" 3 lines, 90 characters
:
```

means you asked edit to fetch the file named "text" for editing, causing it to copy the 90 characters of text into the buffer. Edit awaits your further instructions, and indicates this by its prompt character, the colon (:). In this session, we will append more text to our file, print the contents of the buffer, and learn to change the text of a line.

Adding more text to the file

If you want to add more to the end of your text you may do so by using the append command to enter text input mode. When "append" is the first command of your editing session, the lines you enter are placed at the end of the buffer. Here we'll use the abbreviation for the append command, "a":

```
:a
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
.
```

You may recall that once you enter append mode using the "a" (or "append") command, you need to type a line containing only a period (.) to exit append mode.

Interrupt

Should you press the RUB key (sometimes labelled DELETE) while working with edit, it will send this message to you:

```
Interrupt
:
```

Any command that edit might be executing is terminated by rub or delete, causing edit to prompt you for a new command. If you are appending text at the time, you will exit from append mode and be expected to give another command. The line of text you were typing when the append command was interrupted will not be entered into the buffer.

Making corrections

If while typing the line you hit an incorrect key, recall that you may delete the incorrect character or cancel the entire line of input by erasing in the usual way. Refer either to the last few pages of Session 1 or to "Communicating with UNIX" if you need to review the procedures for making a correction. The most important idea to remember is that erasing a character or cancelling a line must be done before you press the RETURN key.

Listing what's in the buffer (p)

Having appended text to what you wrote in Session 1, you might want to see all the lines in the buffer. To print the contents of the buffer, type the command:

```
:1,$p
```

The "1"† stands for line 1 of the buffer, the "$" is a special symbol designating the last line of the buffer, and "p" (or **print**) is the command to print from line 1 to the end of the buffer. The command "1,$p" gives you:

```
This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.
```

Occasionally, you may accidentally type a character that can't be printed, which can be done by striking a key while the CTRL key is pressed. In printing lines, edit uses a special notation to show the existence of non-printing characters. Suppose you had introduced the non-printing character "control-A" into the word "illustrate" by accidently pressing the CTRL key while typing "a". This can happen on many terminals because the CTRL key and the "A" key are beside each other. If your finger presses between the two keys, control-A results. When asked to print the contents of the buffer, edit would display

```
it does illustr^Ate the editor.
```

To represent the control-A, edit shows "^A". The sequence "^" followed by a capital letter stands for the one character entered by holding down the CTRL key and typing the letter which appears after the "^". We'll soon discuss the commands that can be used to correct this typing error.

In looking over the text we see that "this" is typed as "thiss" in the second line, a deliberate error so we can learn to make corrections. Let's correct the spelling.

Finding things in the buffer

In order to change something in the buffer we first need to find it. We can find "thiss" in the text we have entered by looking at a listing of the lines. Physically speaking, we search the lines of text looking for "thiss" and stop searching when we have found it. The way to tell edit to search for something is to type it inside slash marks:

```
:/thiss/
```

By typing **/thiss/** and pressing RETURN, you instruct edit to search for "thiss". If you ask edit to look for a pattern of characters which it cannot find in the buffer, it will respond "Pattern not found". When edit finds the characters "thiss", it will print the line of text for your inspection:

```
And thiss is some more text.
```

Edit is now positioned in the buffer at the line it just printed, ready to make a change in the line.

†The numeral "one" is the top left-most key, and should not be confused with the letter "el".

The current line

Edit keeps track of the line in the buffer where it is located at all times during an editing session. In general, the line that has been most recently printed, entered, or changed is the current location in the buffer. The editor is prepared to make changes at the current location in the buffer, unless you direct it to another location.

In particular, when you bring a file into the buffer, you will be located at the last line in the file, where the editor left off copying the lines from the file to the buffer. If your first editing command is "append", the lines you enter are added to the end of the file, after the current line — the last line in the file.

You can refer to your current location in the buffer by the symbol period (.) usually known by the name "dot". If you type "." and carriage return you will be instructing edit to print the current line:

```
:.
And thiss is some more text.
```

If you want to know the number of the current line, you can type **.=** and press RETURN, and edit will respond with the line number:

```
:.=
2
```

If you type the number of any line and press RETURN, edit will position you at that line and print its contents:

```
:2
And thiss is some more text.
```

You should experiment with these commands to gain experience in using them to make changes.

Numbering lines (nu)

The **number (nu)** command is similar to print, giving both the number and the text of each printed line. To see the number and the text of the current line type

```
:nu
         2  And thiss is some more text.
```

Note that the shortest abbreviation for the number command is "nu" (and not "n", which is used for a different command). You may specify a range of lines to be listed by the number command in the same way that lines are specified for print. For example, **1,$nu** lists all lines in the buffer with their corresponding line numbers.

Substitute command (s)

Now that you have found the misspelled word, you can change it from "thiss" to "this". As far as edit is concerned, changing things is a matter of substituting one thing for another. As *a* stood for *append,* so *s* stands for *substitute.* We will use the abbreviation "s" to reduce the chance of mistyping the substitute command. This command will instruct edit to make the change:

2s/thiss/this/

We first indicate the line to be changed, line 2, and then type an "s" to indicate we want edit to make a substitution. Inside the first set of slashes are the characters that we want to change, followed by the characters to replace them, and then a closing slash mark. To summarize:

2s/ *what is to be changed* / *what to change it to* /

If edit finds an exact match of the characters to be changed it will make the change **only** in the first occurrence of the characters. If it does not find the characters to be changed, it will respond:

```
Substitute pattern match failed
```

indicating that your instructions could not be carried out. When edit does find the characters that you want to change, it will make the substitution and automatically print the changed line, so that you can check that the correct substitution was made. In the example,

```
:2s/thiss/this/
And this is some more text.
```

line 2 (and line 2 only) will be searched for the characters "thiss", and when the first exact match is found, "thiss" will be changed to "this". Strictly speaking, it was not necessary above to specify the number of the line to be changed. In

```
:s/thiss/this/
```

edit will assume that we mean to change the line where we are currently located ("."). In this case, the command without a line number would have produced the same result because we were already located at the line we wished to change.

For another illustration of the substitute command, let us choose the line:

```
Text editing is strange, but nice.
```

You can make this line a bit more positive by taking out the characters "strange, but " so the line reads:

```
Text editing is nice.
```

A command that will first position edit at the desired line and then make the substitution is:

```
:/strange/s/strange, but //
```

What we have done here is combine our search with our substitution. Such combinations are perfectly legal, and speed up editing quite a bit once you get used to them. That is, you do not necessarily have to use line numbers to identify a line to edit. Instead, you may identify the line you want to change by asking edit to search for a specified pattern of letters that occurs in that line. The parts of the above command are:

/strange/	tells edit to find the characters "strange" in the text
s	tells edit to make a substitution
/strange, but //	substitutes nothing at all for the characters "strange, but "

You should note the space after "but" in "/strange, but /". If you do not indicate that the space is to be taken out, your line will read:

```
Text editing is  nice.
```

which looks a little funny because of the extra space between "is" and "nice". Again, we realize from this that a blank space is a real character to a computer, and in editing text we need to be aware of spaces within a line just as we would be aware of an "a" or a "4".

Another way to list what's in the buffer (z)

Although the print command is useful for looking at specific lines in the buffer, other commands may be more convenient for viewing large sections of text. You can ask to see a screen full of text at a time by using the command **z.** If you type

```
:1z
```

edit will start with line 1 and continue printing lines, stopping either when the screen of your terminal is full or when the last line in the buffer has been printed. If you want to read the next segment of text, type the command

```
:z
```

If no starting line number is given for the z command, printing will start at the "current" line, in this case the last line printed. Viewing lines in the buffer one screen full at a time is known as *paging.* Paging can also be used to print a section of text on a hard-copy terminal.

Saving the modified text

This seems to be a good place to pause in our work, and so we should end the second session. If you (in haste) type "q" to quit the session your dialogue with edit will be:

```
:q
No write since last change (:quit! overrides)
:
```

This is edit's warning that you have not written the modified contents of the buffer to disk. You run the risk of losing the work you did during the editing session since you typed the latest write command. Because in this lesson we have not written to disk at all, everything we have done would have been lost if edit had obeyed the **q** command. If you did not want to save the work done during this editing session, you would have to type "q!" or ("quit!") to confirm that you indeed wanted to end the session immediately, leaving the file as it was after the most recent "write" command. However, since you want to save what you have edited, you need to type:

```
:w
"text" 6 lines, 171 characters
```

and then follow with the commands to quit and logout:

```
:q
% logout
```

and hang up the phone or turn off the terminal when UNIX asks for a name. Terminals connected to the port selector will stop after the logout command, and pressing keys on the keyboard will do nothing.

This is the end of the second session on UNIX text editing.

Session 3

Bringing text into the buffer (e)

Login to UNIX and make contact with edit. You should try to login without looking at the notes, but if you must then by all means do.

Did you remember to give the name of the file you wanted to edit? That is, did you type

```
% edit text
```

or simply

```
% edit
```

Both ways get you in contact with edit, but the first way will bring a copy of the file named "text" into the buffer. If you did forget to tell edit the name of your file, you can get it into the buffer by typing:

```
:e text
"text" 6 lines, 171 characters
```

The command **edit,** which may be abbreviated e, tells edit that you want to erase anything that might already be in the buffer and bring a copy of the file "text" into the buffer for editing. You may also use the edit (e) command to change files in the middle of an editing session, or to give edit the name of a new file that you want to create. Because the edit command clears the buffer, you will receive a warning if you try to edit a new file without having saved a copy of the old file. This gives you a chance to write the contents of the buffer to disk before editing the next file.

Moving text in the buffer (m)

Edit allows you to move lines of text from one location in the buffer to another by means of the **move (m)** command. The first two examples are for illustration only, though after you have read this Session you are welcome to return to them for practice. The command

```
:2,4m$
```

directs edit to move lines 2, 3, and 4 to the end of the buffer ($). The format for the move command is that you specify the first line to be moved, the last line to be moved, the move command "m", and the line after which the moved text is to be placed. So,

```
:1,3m6
```

would instruct edit to move lines 1 through 3 (inclusive) to a location after line 6 in the buffer. To move only one line, say, line 4, to a location in the buffer after line 5, the command would be "4m5".

Let's move some text using the command:

```
:5,$m1
2 lines moved
it does illustrate the editor.
```

After executing a command that moves more than one line of the buffer, edit tells how many lines were affected by the move and prints the last moved line for your inspection. If you want to see more than just the last line, you can then use the print (p), z, or number (nu) command to view more text. The buffer should now contain:

```
This is some sample text.
It doesn't mean much here, but
it does illustrate the editor.
And this is some more text.
Text editing is nice.
This is text added in Session 2.
```

You can restore the original order by typing:

```
:4,$m1
```

or, combining context searching and the move command:

```
:/And this is some/,/This is text/m/This is some sample/
```

(Do not type both examples here!) The problem with combining context searching with the move command is that your chance of making a typing error in such a long command is greater than if you type line numbers.

Copying lines (copy)

The **copy** command is used to make a second copy of specified lines, leaving the original lines where they were. Copy has the same format as the move command, for example:

```
:2,5copy $
```

makes a copy of lines 2 through 5, placing the added lines after the buffer's end ($). Experiment with the copy command so that you can become familiar with how it works. Note that the shortest abbreviation for copy is **co** (and not the letter "c", which has another meaning).

Deleting lines (d)

Suppose you want to delete the line

```
This is text added in Session 2.
```

from the buffer. If you know the number of the line to be deleted, you can type that number followed by **delete** or **d**. This example deletes line 4, which is "This is text added in Session 2." if you typed the commands suggested so far.

```
:4d
It doesn't mean much here, but
```

Here "4" is the number of the line to be deleted, and "delete" or "d" is the command to delete the line. After executing the delete command, edit prints the line that has become the current line (".").

If you do not happen to know the line number you can search for the line and then delete it using this sequence of commands:

```
:/added in Session 2./
This is text added in Session 2.
:d
It doesn't mean much here, but
```

The "/added in Session 2./" asks edit to locate and print the line containing the indicated text, starting its search at the current line and moving line by line until it finds the text. Once you are sure that you have correctly specified the line you want to delete, you can enter the delete (d) command. In this case it is not necessary to specify a line number before the "d". If no line number is given, edit deletes the current line ("."), that is, the line found by our search. After the deletion, your buffer should contain:

```
This is some sample text.
And this is some more text.
Text editing is nice.
It doesn't mean much here, but
it does illustrate the editor.
And this is some more text.
Text editing is nice.
This is text added in Session 2.
It doesn't mean much here, but
```

To delete both lines 2 and 3:

```
And this is some more text.
Text editing is nice.
```

you type

```
:2,3d
2 lines deleted
```

which specifies the range of lines from 2 to 3, and the operation on those lines — "d" for delete. If you delete more than one line you will receive a message telling you the number of lines deleted, as indicated in the example above.

The previous example assumes that you know the line numbers for the lines to be deleted. If you do not you might combine the search command with the delete command:

```
:/And this is some/,/Text editing is nice./d
```

A word or two of caution

In using the search function to locate lines to be deleted you should be **absolutely sure** the characters you give as the basis for the search will take edit to the line you want deleted. Edit will search for the first occurrence of the characters starting from where you last edited — that is, from the line you see printed if you type dot (.).

A search based on too few characters may result in the wrong lines being deleted, which edit will do as easily as if you had meant it. For this reason, it is usually safer to specify the search and then delete in two separate steps, at least until you become familiar enough with using the editor that you understand how best to specify searches. For a beginner it is not a bad idea to double-check each command before pressing RETURN to send the command on its way.

Undo (u) to the rescue

The **undo (u)** command has the ability to reverse the effects of the last command that changed the buffer. To undo the previous command, type "u" or "undo". Undo can rescue the contents of the buffer from many an unfortunate mistake. However, its powers are not unlimited, so it is still wise to be reasonably careful about the commands you give.

It is possible to undo only commands which have the power to change the buffer — for example, delete, append, move, copy, substitute, and even undo itself. The commands write (w) and edit (e), which interact with disk files, cannot be undone, nor can commands that do not change the buffer, such as print. Most importantly, the **only** command that can be reversed by undo is the last "undo-able" command you typed. You can use control-H and @ to change commands while you are typing them, and undo to reverse the effect of the commands after you have typed them and pressed RETURN.

To illustrate, let's issue an undo command. Recall that the last buffer-changing command we gave deleted the lines formerly numbered 2 and 3. Typing undo at this moment will reverse the effects of the deletion, causing those two lines to be replaced in the buffer.

```
:u
2 more lines in file after undo
And this is some more text.
```

Here again, edit informs you if the command affects more than one line, and prints the text of the line which is now "dot" (the current line).

More about the dot (.) and buffer end ($)

The function assumed by the symbol dot depends on its context. It can be used:

1. to exit from append mode; we type dot (and only a dot) on a line and press RETURN;
2. to refer to the line we are at in the buffer.

Dot can also be combined with the equal sign to get the number of the line currently being edited:

```
:.=
```

If we type ".=" we are asking for the number of the line, and if we type "." we are asking for the text of the line.

In this editing session and the last, we used the dollar sign to indicate the end of the buffer in commands such as print, copy, and move. The dollar sign as a command asks edit to print the last line in the buffer. If the dollar sign is combined with the equal sign (**$=**) edit will print the line number corresponding to the last line in the buffer.

"." and "$", then, represent line numbers. Whenever appropriate, these symbols can be used in place of line numbers in commands. For example

```
:.,$d
```

instructs edit to delete all lines from the current line (.) to the end of the buffer.

Moving around in the buffer (+ and −)

When you are editing you often want to go back and re-read a previous line. You could specify a context search for a line you want to read if you remember some of its text, but if you simply want to see what was written a few, say 3, lines ago, you can type

```
−3p
```

This tells edit to move back to a position 3 lines before the current line (.) and print that line. You can move forward in the buffer similarly:

```
+2p
```

instructs edit to print the line that is 2 ahead of your current position.

You may use "+" and "−" in any command where edit accepts line numbers. Line numbers specified with "+" or "−" can be combined to print a range of lines. The command

```
:−1,+2copy$
```

makes a copy of 4 lines: the current line, the line before it, and the two after it. The copied lines will be placed after the last line in the buffer ($), and the original lines referred to by "−1" and "+2" remain where they are.

Try typing only "−"; you will move back one line just as if you had typed "−1p". Typing the command "+" works similarly. You might also try typing a few plus or minus signs in a row (such as "+++") to see edit's response. Typing RETURN alone on a line is the equivalent of typing "+1p"; it will move you one line ahead in the buffer and print that line.

If you are at the last line of the buffer and try to move further ahead, perhaps by typing a "+" or a carriage return alone on the line, edit will remind you that you are at the end of the buffer:

```
        At end-of-file
```

or

```
        Not that many lines in buffer
```

Similarly, if you try to move to a position before the first line, edit will print one of these messages:

```
        Nonzero address required on this command
```

or

```
        Negative address — first buffer line is 1
```

The number associated with a buffer line is the line's "address", in that it can be used to locate the line.

Changing lines (c)

You can also delete certain lines and insert new text in their place. This can be accomplished easily with the **change (c)** command. The change command instructs edit to delete specified lines and then switch to text input mode to accept the text that will replace them. Let's say you want to change the first two lines in the buffer:

```
        This is some sample text.
        And this is some more text.
```

to read

```
        This text was created with the UNIX text editor.
```

To do so, you type:

```
        :1,2c
        2 lines changed
        This text was created with the UNIX text editor.
        .
        :
```

In the command **1,2c** we specify that we want to change the range of lines beginning with 1 and ending with 2 by giving line numbers as with the print command. These lines will be deleted. After you type RETURN to end the change command, edit notifies you if more than one line will be changed and places you in text input mode. Any text typed on the following lines will be inserted into the position where lines were deleted by the change command. **You will remain in text input mode until you exit in the usual way, by typing a period alone on a line.** Note that the number of lines added to the buffer need not be the same as the number of lines deleted.

This is the end of the third session on text editing with UNIX.

Session 4

This lesson covers several topics, starting with commands that apply throughout the buffer, characters with special meanings, and how to issue UNIX commands while in the editor. The next topics deal with files: more on reading and writing, and methods of recovering files lost in a crash. The final section suggests sources of further information.

Making commands global (g)

One disadvantage to the commands we have used for searching or substituting is that if you have a number of instances of a word to change it appears that you have to type the command repeatedly, once for each time the change needs to be made. Edit, however, provides a way to make commands apply to the entire contents of the buffer — the **global (g)** command.

To print all lines containing a certain sequence of characters (say, "text") the command is:

```
:g/text/p
```

The "g" instructs edit to make a global search for all lines in the buffer containing the characters "text". The "p" prints the lines found.

To issue a global command, start by typing a "g" and then a search pattern identifying the lines to be affected. Then, on the same line, type the command to be executed for the identified lines. Global substitutions are frequently useful. For example, to change all instances of the word "text" to the word "material" the command would be a combination of the global search and the substitute command:

```
:g/text/s/text/material/g
```

Note the "g" at the end of the global command, which instructs edit to change each and every instance of "text" to "material". If you do not type the "g" at the end of the command only the *first* instance of "text" *in each line* will be changed (the normal result of the substitute command). The "g" at the end of the command is independent of the "g" at the beginning. You may give a command such as:

```
:5s/text/material/g
```

to change every instance of "text" in line 5 alone. Further, neither command will change "text" to "material" if "Text" begins with a capital rather than a lower-case *t.*

Edit does not automatically print the lines modified by a global command. If you want the lines to be printed, type a "p" at the end of the global command:

```
:g/text/s/text/material/gp
```

You should be careful about using the global command in combination with any other — in essence, be sure of what you are telling edit to do to the entire buffer. For example,

```
:g/ /d
72 less lines in file after global
```

will delete every line containing a blank anywhere in it. This could adversely affect your document, since most lines have spaces between words and thus would be deleted. After executing the global command, edit will print a warning if the command added or deleted more than one line. Fortunately, the undo command can reverse the effects of a global command. You should experiment with the global command on a small file of text to see what it can do for you.

More about searching and substituting

In using slashes to identify a character string that we want to search for or change, we have always specified the exact characters. There is a less tedious way to repeat the same string of characters. To change "text" to "texts" we may type either

```
:/text/s/text/texts/
```

as we have done in the past, or a somewhat abbreviated command:

```
:/text/s//texts/
```

In this example, the characters to be changed are not specified — there are no characters, not even a space, between the two slash marks that indicate what is to be changed. This lack of characters between the slashes is taken by the editor to mean "use the characters we last searched for as the characters to be changed."

Similarly, the last context search may be repeated by typing a pair of slashes with nothing between them:

```
:/does/
It doesn't mean much here, but
://
it does illustrate the editor.
```

(You should note that the search command found the characters "does" in the word "doesn't" in the first search request.) Because no characters are specified for the second search, the editor scans the buffer for the next occurrence of the characters "does".

Edit normally searches forward through the buffer, wrapping around from the end of the buffer to the beginning, until the specified character string is found. If you want to search in the reverse direction, use question marks (?) instead of slashes to surround the characters you are searching for.

It is also possible to repeat the last substitution without having to retype the entire command. An ampersand (&) used as a command repeats the most recent substitute command, using the same search and replacement patterns. After altering the current line by typing

```
:s/text/texts/
```

you type

```
:/text/&
```

or simply

```
://&
```

to make the same change on the next line in the buffer containing the characters "text".

Special characters

Two characters have special meanings when used in specifying searches: "$" and "^". "$" is taken by the editor to mean "end of the line" and is used to identify strings that occur at the end of a line.

```
:g/text.$/s//material./p
```

tells the editor to search for all lines ending in "text." (and nothing else, not even a blank space), to change each final "text." to "material.", and print the changed lines.

The symbol "^" indicates the beginning of a line. Thus,

```
:s/^/1. /
```

instructs the editor to insert "1." and a space at the beginning of the current line.

The characters "$" and "^" have special meanings only in the context of searching. At other times, they are ordinary characters. If you ever need to search for a character that has a special meaning, you must indicate that the character is to lose temporarily its special significance by typing another special character, the backslash (\), before it.

```
:s/\$/dollar/
```

looks for the character "$" in the current line and replaces it by the word "dollar". Were it not for the backslash, the "$" would have represented "the end of the line" in your search rather than the character "$". The backslash retains its special significance unless it is preceded by another backslash.

Issuing UNIX commands from the editor

After creating several files with the editor, you may want to delete files no longer useful to you or ask for a list of your files. Removing and listing files are not functions of the editor, and so they require the use of UNIX system commands (also referred to as "shell" commands, as "shell" is the name of the program that processes UNIX commands). You do not need to quit the editor to execute a UNIX command as long as you indicate that it is to be sent to the shell for execution. To use the UNIX command **rm** to remove the file named "junk" type:

```
:!rm junk
!
:
```

The exclamation mark (!) indicates that the rest of the line is to be processed as a shell command. If the buffer contents have not been written since the last change, a warning will be printed before the command is executed:

```
[No write since last change]
```

The editor prints a "!" when the command is completed. The tutorial "Communicating with UNIX" describes useful features of the system, of which the editor is only one part.

Filenames and file manipulation

Throughout each editing session, edit keeps track of the name of the file being edited as the *current filename.* Edit remembers as the current filename the name given when you entered the editor. The current filename changes whenever the edit (e) command is used to specify a new file. Once edit has recorded a current filename, it inserts that name into any command where a filename has been omitted. If a write command does not specify a file, edit, as we have seen, supplies the current filename. If you are editing a file named "draft3" having 283 lines in it, you can have the editor write onto a different file by including its name in the write command:

```
:w chapter3
"chapter3" [new file] 283 lines, 8698 characters
```

The current filename remembered by the editor *will not be changed as a result of the write command.* Thus, if the next write command does not specify a name, edit will write onto the current file ("draft3") and not onto the file "chapter3".

The file (f) command

To ask for the current filename, type **file** (or **f**). In response, the editor provides current information about the buffer, including the filename, your current position, the number of lines in the buffer, and the percent of the distance through the file your current location is.

```
:f
"text" [Modified] line 3 of 4 --75%--
```

If the contents of the buffer have changed since the last time the file was written, the editor will tell you that the file has been "[Modified]". After you save the changes by writing onto a disk file, the buffer will no longer be considered modified:

```
:w
"text" 4 lines, 88 characters
:f
"text" line 3 of 4 --75%--
```

Reading additional files (r)

The **read (r)** command allows you to add the contents of a file to the buffer at a specified location, essentially copying new lines between two existing lines. To use it, specify the line after which the new text will be placed, the **read (r)** command, and then the name of the file. If you have a file named "example", the command

```
:$r example
"example" 18 lines, 473 characters
```

reads the file "example" and adds it to the buffer after the last line. The current filename is not changed by the read command.

Writing parts of the buffer

The **write (w)** command can write all or part of the buffer to a file you specify. We are already familiar with writing the entire contents of the buffer to a disk file. To write only part of the buffer onto a file, indicate the beginning and ending lines before the write command, for example

```
:45,$w ending
```

Here all lines from 45 through the end of the buffer are written onto the file named *ending*. The lines remain in the buffer as part of the document you are editing, and you may continue to edit the entire buffer. Your original file is unaffected by your command to write part of the buffer to another file. Edit still remembers whether you have saved changes to the buffer in your original file or not.

Recovering files

Although it does not happen very often, there are times UNIX stops working because of some malfunction. This situation is known as a *crash*. Under most circumstances, edit's crash recovery feature is able to save work to within a few lines of changes before a crash (or an accidental phone hang up). If you lose the contents of an editing buffer in a system crash, you will normally receive mail when you login that gives the name of the recovered file. To recover the file, enter the editor and type the command **recover (rec)**, followed by the name of the lost file. For example, to recover the buffer for an edit session involving the file "chap6", the command is:

```
:recover chap6
```

Recover is sometimes unable to save the entire buffer successfully, so always check the contents of the saved buffer carefully before writing it back onto the original file. For best results, write the buffer to a new file temporarily so you can examine it without risk to the original file. Unfortunately, you cannot use the recover command to retrieve a file you removed using the shell command **rm**.

Other recovery techniques

If something goes wrong when you are using the editor, it may be possible to save your work by using the command **preserve (pre)**, which saves the buffer as if the system had crashed. If you are writing a file and you get the message "Quota exceeded", you have tried to use more disk storage than is allotted to your account. *Proceed with caution* because it is likely that only a part of the editor's buffer is now present in the file you tried to write. In this case you should use the shell escape from the editor (!) to remove some files you don't need and try to write the file again. If this is not possible and you cannot find someone to help you, enter the command

```
:preserve
```

and wait for the reply,

```
File preserved.
```

If you do not receive this reply, seek help immediately. Do not simply leave the editor. If you do, the buffer will be lost, and you may not be able to save your file. If the reply is "File preserved." you can leave the editor (or logout) to remedy the situation. After a preserve, you can use the recover command once the problem has been corrected, or the **−r** option of the edit command if you leave the editor and want to return.

If you make an undesirable change to the buffer and type a write command before discovering your mistake, the modified version will replace any previous version of the file. Should you ever lose a good version of a document in this way, do not panic and leave the editor. As long as you stay in the editor, the contents of the buffer remain accessible. Depending on the nature of the problem, it may be possible to restore the buffer to a more complete state with the undo command. After fixing the damaged buffer, you can again write the file to disk.

Further reading and other information

Edit is an editor designed for beginning and casual users. It is actually a version of a more powerful editor called *ex.* These lessons are intended to introduce you to the editor and its more commonly-used commands. We have not covered all of the editor's commands, but a selection of commands that should be sufficient to accomplish most of your editing tasks. You can find out more about the editor in the *Ex Reference Manual,* which is applicable to both *ex* and *edit.* The manual is available from the Computing Services Library, 218 Evans Hall. One way to become familiar with the manual is to begin by reading the description of commands that you already know.

Using *ex*

As you become more experienced with using the editor, you may still find that edit continues to meet your needs. However, should you become interested in using ex, it is easy to switch. To begin an editing session with ex, use the name **ex** in your command instead of **edit.**

Edit commands work the same way in ex, but the editing environment is somewhat different. You should be aware of a few differences that exist between the two versions of the editor. In edit, only the characters "^", "$", and "\" have special meanings in searching the buffer or indicating characters to be changed by a substitute command. Several additional characters have special meanings in ex, as described in the *Ex Reference Manual.* Another feature of the edit environment prevents users from accidently entering two alternative modes of editing, *open* and *visual,* in which the editor behaves quite differently from normal command mode. If you are using ex and the editor behaves strangely, you may have accidently entered open mode by typing "o". Type the ESC key and then a "Q" to get out of open or visual mode and back into the regular editor command mode. The document *An Introduction to Display Editing with Vi* provides a full discussion of visual mode.

Index

An Introduction to Display Editing with Vi

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, Ca. 94720

ABSTRACT

Vi (visual) is a display oriented interactive text editor. When using *vi* the screen of your terminal acts as a window into the file which you are editing. Changes which you make to the file are reflected in what you see.

Using *vi* you can insert new text any place in the file quite easily. Most of the commands to *vi* move the cursor around in the file. There are commands to move the cursor forward and backward in units of characters, words, sentences and paragraphs. A small set of operators, like **d** for delete and **c** for change, are combined with the motion commands to form operations such as delete word or change paragraph, in a simple and natural way. This regularity and the mnemonic assignment of commands to keys makes the editor command set easy to remember and to use..

Vi will work on a large number of display terminals, and new terminals are easily driven after editing a terminal description file. While it is advantageous to have an intelligent terminal which can locally insert and delete lines and characters from the display, the editor will function quite well on dumb terminals over slow phone lines. The editor makes allowance for the low bandwidth in these situations and uses smaller window sizes and different display updating algorithms to make best use of the limited speed available.

It is also possible to use the command set of *vi* on hardcopy terminals, storage tubes and "glass tty's" using a one line editing window; thus *vi's* command set is available on all terminals. The full command set of the more traditional, line oriented editor *ex* is available within *vi;* it is quite simple to switch between the two modes of editing.

September 16, 1980

An Introduction to Display Editing with Vi

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, Ca. 94720

1. Getting started

This document provides a quick introduction to *vi.* (Pronounced *vee-eye.*) You should be running *vi* on a file you are familiar with while you are reading this. The first part of this document (sections 1 through 5) describes the basics of using *vi.* Some topics of special interest are presented in section 6, and some nitty-gritty details of how the editor functions are saved for section 7 to avoid cluttering the presentation here.

There is also a short appendix here, which gives for each character the special meanings which this character has in *vi.* Attached to this document should be a quick reference card. This card summarizes the commands of *vi* in a very compact format. You should have the card handy while you are learning *vi.*

1.1. Specifying terminal type

Before you can start *vi* you must tell the system what kind of terminal you are using. Here is a (necessarily incomplete) list of terminal type codes. If your terminal does not appear here, you should consult with one of the staff members on your system to find out the code for your terminal. If your terminal does not have a code, one can be assigned and a description for the terminal can be created.

Code	Full name	Type
2621	Hewlett-Packard 2621A/P	Intelligent
2645	Hewlett-Packard 264x	Intelligent
act4	Microterm ACT-IV	Dumb
act5	Microterm ACT-V	Dumb
adm3a	Lear Siegler ADM-3a	Dumb
adm31	Lear Siegler ADM-31	Intelligent
c100	Human Design Concept 100	Intelligent
dm1520	Datamedia 1520	Dumb
dm2500	Datamedia 2500	Intelligent
dm3025	Datamedia 3025	Intelligent
fox	Perkin-Elmer Fox	Dumb
h1500	Hazeltine 1500	Intelligent
h19	Heathkit h19	Intelligent
i100	Infoton 100	Intelligent
mime	Imitating a smart act4	Intelligent

The financial support of an IBM Graduate Fellowship and the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 is gratefully acknowledged.

t1061	Teleray 1061	Intelligent
vt52	Dec VT-52	Dumb

Suppose for example that you have a Hewlett-Packard HP2621A terminal. The code used by the system for this terminal is '2621'. In this case you can use one of the following commands to tell the system the type of your terminal:

```
% setenv TERM 2621
```

This command works with the shell *csh* on both version 6 and 7 systems. If you are using the standard version 7 shell then you should give the commands

```
$ TERM=2621
$ export TERM
```

If you want to arrange to have your terminal type set up automatically when you log in, you can use the *tset* program. If you dial in on a *mime*, but often use hardwired ports, a typical line for your *.login* file (if you use csh) would be

```
setenv TERM `tset - -d mime`
```

or for your *.profile* file (if you use sh)

```
TERM=`tset - -d mime`
```

Tset knows which terminals are hardwired to each port and needs only to be told that when you dial in you are probably on a *mime*. *Tset* is usually used to change the erase and kill characters, too.

1.2. Editing a file

After telling the system which kind of terminal you have, you should make a copy of a file you are familiar with, and run *vi* on this file, giving the command

```
% vi name
```

replacing *name* with the name of the copy file you just created. The screen should clear and the text of your file should appear on the screen. If something else happens refer to the footnote.‡

1.3. The editor's copy: the buffer

The editor does not directly modify the file which you are editing. Rather, the editor makes a copy of this file, in a place called the *buffer*, and remembers the file's name. You do not affect the contents of the file unless and until you write the changes you make back into the original file.

‡ If you gave the system an incorrect terminal type code then the editor may have just made a mess out of your screen. This happens when it sends control codes for one kind of terminal to some other kind of terminal. In this case hit the keys :q (colon and the q key) and then hit the RETURN key. This should get you back to the command level interpreter. Figure out what you did wrong (ask someone else if necessary) and try again.

Another thing which can go wrong is that you typed the wrong file name and the editor just printed an error diagnostic. In this case you should follow the above procedure for getting out of the editor, and try again this time spelling the file name correctly.

If the editor doesn't seem to respond to the commands which you type here, try sending an interrupt to it by hitting the DEL or RUB key on your terminal, and then hitting the :q command again followed by a carriage return.

1.4. Notational conventions

In our examples, input which must be typed as is will be presented in **bold face**. Text which should be replaced with appropriate input will be given in *italics*. We will represent special characters in SMALL CAPITALS.

1.5. Arrow keys

The editor command set is independent of the terminal you are using. On most terminals with cursor positioning keys, these keys will also work within the editor. If you don't have cursor positioning keys, or even if you do, you can use the **h j k** and **l** keys as cursor positioning keys (these are labelled with arrows on an *adm3a).**

(Particular note for the HP2621: on this terminal the function keys must be *shifted* (ick) to send to the machine, otherwise they only act locally. Unshifted use will leave the cursor positioned incorrectly.)

1.6. Special characters: ESC, CR and DEL

Several of these special characters are very important, so be sure to find them right now. Look on your keyboard for a key labelled ESC or ALT. It should be near the upper left corner of your terminal. Try hitting this key a few times. The editor will ring the bell to indicate that it is in a quiescent state.‡ Partially formed commands are cancelled by ESC, and when you insert text in the file you end the text insertion with ESC. This key is a fairly harmless one to hit, so you can just hit it if you don't know what is going on until the editor rings the bell.

The CR or RETURN key is important because it is used to terminate certain commands. It is usually at the right side of the keyboard, and is the same command used at the end of each shell command.

Another very useful key is the DEL or RUB key, which generates an interrupt, telling the editor to stop what it is doing. It is a forceful way of making the editor listen to you, or to return it to the quiescent state if you don't know or don't like what is going on. Try hitting the '**/**' key on your terminal. This key is used when you want to specify a string to be searched for. The cursor should now be positioned at the bottom line of the terminal after a '/' printed as a prompt. You can get the cursor back to the current position by hitting the DEL or RUB key; try this now.* From now on we will simply refer to hitting the DEL or RUB key as "sending an interrupt."**

The editor often echoes your commands on the last line of the terminal. If the cursor is on the first position of this last line, then the editor is performing a computation, such as computing a new position in the file after a search or running a command to reformat part of the buffer. When this is happening you can stop the editor by sending an interrupt.

1.7. Getting out of the editor

After you have worked with this introduction for a while, and you wish to do something else, you can give the command **ZZ** to the editor. This will write the contents of the editor's buffer back into the file you are editing, if you made any changes, and then quit from the editor. You can also end an editor session by giving the command **:q!**CR;† this is a dangerous but occasionally essential command which ends the editor session and discards all your changes. You need to know about this command in case you change the editor's copy of a file you wish

---

* As we will see later, *h* moves back to the left (like control-h which is a backspace), *j* moves down (in the same column), *k* moves up (in the same column), and *l* moves to the right.

‡ On smart terminals where it is possible, the editor will quietly flash the screen rather than ringing the bell.

* Backspacing over the '/' will also cancel the search.

** On some systems, this interruptibility comes at a price: you cannot type ahead when the editor is computing with the cursor on the bottom line.

† All commands which read from the last display line can also be terminated with a ESC as well as an CR.

only to look at. Be very careful not to give this command when you really want to save the changes you have made.

2. Moving around in the file

2.1. Scrolling and paging

The editor has a number of commands for moving around in the file. The most useful of these is generated by hitting the control and D keys at the same time, a control-D or '^D'. We will use this two character notation for referring to these control keys from now on. You may have a key labelled '^' on your terminal. This key will be represented as '↑' in this document; '^' is exclusively used as part of the '^x' notation for control characters.‡

As you know now if you tried hitting ^D, this command scrolls down in the file. The D thus stands for down. Many editor commands are mnemonic and this makes them much easier to remember. For instance the command to scroll up is ^U. Many dumb terminals can't scroll up at all, in which case hitting ^U clears the screen and refreshes it with a line which is farther back in the file at the top.

If you want to see more of the file below where you are, you can hit ^E to expose one more line at the bottom of the screen, leaving the cursor where it is. ‡‡ The command ^Y (which is hopelessly non-mnemonic, but next to ^U on the keyboard) exposes one more line at the top of the screen.

There are other ways to move around in the file; the keys ^F and ^B ‡ move forward and backward a page, keeping a couple of lines of continuity between screens so that it is possible to read through a file using these rather than ^D and ^U if you wish.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, hitting ^F to move forward a page will leave you only a little context to look back at. Scrolling on the other hand leaves more context, and happens more smoothly. You can continue to read the text as scrolling is taking place.

2.2. Searching, goto, and previous context

Another way to position yourself in the file is by giving the editor a string to search for. Type the character / followed by a string of characters terminated by CR. The editor will position the cursor at the next occurrence of this string. Try hitting **n** to then go to the next occurrence of this string. The character ? will search backwards from where you are, and is otherwise like /.†

If the search string you give the editor is not present in the file the editor will print a diagnostic on the last line of the screen, and the cursor will be returned to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with an ↑. To match only at the end of a line, end the search string with a $. Thus **/↑search**CR will search for the word 'search' at the beginning of a line, and **/last$**CR searches for the word 'last' at the end of a line.*

---

‡ If you don't have a '^' key on your terminal then there is probably a key labelled '↑'; in any case these characters are one and the same.

‡‡ Version 3 only.

‡ Not available in all v2 editors due to memory constraints.

† These searches will normally wrap around the end of the file, and thus find the string even if it is not on a line in the direction you search provided it is anywhere else in the file. You can disable this wraparound in scans by giving the command **:se nowrapscan**CR, or more briefly **:se nows**CR.

*Actually, the string you give to search for here can be a *regular expression* in the sense of the editors *ex*(1) and *ed*(1). If you don't wish to learn about this yet, you can disable this more general facility by doing **:se nomagic**CR; by putting this command in EXINIT in your environment, you can have this always be in effect (more about *EXINIT* later.)

The command G, when preceded by a number will position the cursor at that line in the file. Thus 1G will move the cursor to the first line of the file. If you give G no count, then it moves to the end of the file.

If you are near the end of the file, and the last line is not at the bottom of the screen, the editor will place only the character '~' on each remaining line. This indicates that the last line in the file is on the screen; that is, the '~' lines are past the end of the file.

You can find out the state of the file you are editing by typing a ^G. The editor will show you the name of the file you are editing, the number of the current line, the number of lines in the buffer, and the percentage of the way through the buffer which you are. Try doing this now, and remember the number of the line you are on. Give a G command to get to the end and then another G command to get back where you were.

You can also get back to a previous position by using the command `` (two back quotes). This is often more convenient than G because it requires no advance preparation. Try giving a G or a search with / or ? and then a `` to get back to where you were. If you accidentally hit n or any command which moves you far away from a context of interest, you can quickly get back by hitting ``.

2.3. Moving around on the screen

Now try just moving the cursor around on the screen. If your terminal has arrow keys (4 or 5 keys with arrows going in each direction) try them and convince yourself that they work. (On certain terminals using v2 editors, they won't.) If you don't have working arrow keys, you can always use **h**, **j**, **k**, and **l**. Experienced users of *vi* prefer these keys to arrow keys, because they are usually right underneath their fingers.

Hit the + key. Each time you do, notice that the cursor advances to the next line in the file, at the first non-white position on the line. The − key is like + but goes the other way.

These are very common keys for moving up and down lines in the file. Notice that if you go off the bottom or top with these keys then the screen will scroll down (and up if possible) to bring a line at a time into view. The RETURN key has the same effect as the + key.

Vi also has commands to take you to the top, middle and bottom of the screen. H will take you to the top (home) line on the screen. Try preceding it with a number as in 3H. This will take you to the third line on the screen. Many *vi* commands take preceding numbers and do interesting things with them. Try M, which takes you to the middle line on the screen, and L, which takes you to the last line on the screen. L also takes counts, thus 5L will take you to the fifth line from the bottom.

2.4. Moving within a line

Now try picking a word on some line on the screen, not the first word on the line. move the cursor using RETURN and − to be on the line where the word is. Try hitting the w key. This will advance the cursor to the next word on the line. Try hitting the **b** key to back up words in the line. Also try the e key which advances you to the end of the current word rather than to the beginning of the next word. Also try SPACE (the space bar) which moves right one character and the BS (backspace or ^H) key which moves left one character. The key **h** works as ^H does and is useful if you don't have a BS key. (Also, as noted just above, **l** will move to the right.)

If the line had punctuation in it you may have noticed that that the **w** and **b** keys stopped at each group of punctuation. You can also go back and forwards words without stopping at punctuation by using **W** and **B** rather than the lower case equivalents. Think of these as bigger words. Try these on a few lines with punctuation to see how they differ from the lower case **w** and **b**.

The word keys wrap around the end of line, rather than stopping at the end. Try moving to a word on a line below where you are by repeatedly hitting w.

2.5. Summary

SPACE	advance the cursor one position
^B	backwards to previous page
^D	scrolls down in the file
^E	exposes another line at the bottom (v3)
^F	forward to next page
^G	tell what is going on
^H	backspace the cursor
^N	next line, same column
^P	previous line, same column
^U	scrolls up in the file
^Y	exposes another line at the top (v3)
+	next line, at the beginning
−	previous line, at the beginning
/	scan for a following string forwards
?	scan backwards
B	back a word, ignoring punctuation
G	go to specified line, last default
H	home screen line
M	middle screen line
L	last screen line
W	forward a word, ignoring punctuation
b	back a word
e	end of current word
n	scan for next instance of / or ? pattern
w	word after this word

2.6. View ‡

If you want to use the editor to look at a file, rather than to make changes, invoke it as *view* instead of *vi*. This will set the *readonly* option which will prevent you from accidently overwriting the file.

3. Making simple changes

3.1. Inserting

One of the most useful commands is the **i** (insert) command. After you type **i**, everything you type until you hit ESC is inserted into the file. Try this now; position yourself to some word in the file and try inserting text before this word. If you are on an dumb terminal it will seem, for a minute, that some of the characters in your line have been overwritten, but they will reappear when you hit ESC.

Now try finding a word which can, but does not, end in an 's'. Position yourself at this word and type **e** (move to end of word), then **a** for append and then 'sESC' to terminate the textual insert. This sequence of commands can be used to easily pluralize a word.

Try inserting and appending a few times to make sure you understand how this works; **i** placing text to the left of the cursor, **a** to the right.

It is often the case that you want to add new lines to the file you are editing, before or after some specific line in the file. Find a line where this makes sense and then give the command **o** to create a new line after the line you are on, or the command **O** to create a new line before the line you are on. After you create a new line in this way, text you type up to an ESC

‡ Not available in all v2 editors due to memory constraints.

is inserted on the new line.

Many related editor commands are invoked by the same letter key and differ only in that one is given by a lower case key and the other is given by an upper case key. In these cases, the upper case key often differs from the lower case key in its sense of direction, with the upper case key working backward and/or up, while the lower case key moves forward and/or down.

Whenever you are typing in text, you can give many lines of input or just a few characters. To type in more than one line of text, hit a RETURN at the middle of your input. A new line will be created for text, and you can continue to type. If you are on a slow and dumb terminal the editor may choose to wait to redraw the tail of the screen, and will let you type over the existing screen lines. This avoids the lengthy delay which would occur if the editor attempted to keep the tail of the screen always up to date. The tail of the screen will be fixed up, and the missing lines will reappear, when you hit ESC.

While you are inserting new text, you can use the characters you normally use at the system command level (usually ^H or #) to backspace over the last character which you typed, and the character which you use to kill input lines (usually @, ^X, or ^U) to erase the input you have typed on the current line.† The character ^W will erase a whole word and leave you after the space after the previous word; it is useful for quickly backing up in an insert.

Notice that when you backspace during an insertion the characters you backspace over are not erased; the cursor moves backwards, and the characters remain on the display. This is often useful if you are planning to type in something similar. In any case the characters disappear when when you hit ESC; if you want to get rid of them immediately, hit an ESC and then a again.

Notice also that you can't erase characters which you didn't insert, and that you can't backspace around the end of a line. If you need to back up to the previous line to make a correction, just hit ESC and move the cursor back to the previous line. After making the correction you can return to where you were and use the insert or append command again.

3.2. Making small corrections

You can make small corrections in existing text quite easily. Find a single character which is wrong or just pick any character. Use the arrow keys to find the character, or get near the character with the word motion keys and then either backspace (hit the BS key or ^H or even just **h**) or SPACE (using the space bar) until the cursor is on the character which is wrong. If the character is not needed then hit the **x** key; this deletes the character from the file. It is analogous to the way you x out characters when you make mistakes on a typewriter (except it's not as messy).

If the character is incorrect, you can replace it with the correct character by giving the command *rc*, where *c* is replaced by the correct character. Finally if the character which is incorrect should be replaced by more than one character, give the command **s** which substitutes a string of characters, ending with ESC, for it. If there are a small number of characters which are wrong you can precede **s** with a count of the number of characters to be replaced. Counts are also useful with **x** to specify the number of characters to be deleted.

3.3. More corrections: operators

You already know almost enough to make changes at a higher level. All you need to know now is that the **d** key acts as a delete operator. Try the command **dw** to delete a word. Try hitting **.** a few times. Notice that this repeats the effect of the **dw**. The command **.** repeats the last command which made a change. You can remember it by analogy with an ellipsis '**...**'.

† In fact, the character ^H (backspace) always works to erase the last input character here, regardless of what your erase character is.

Now try **db**. This deletes a word backwards, namely the preceding word. Try **d**SPACE. This deletes a single character, and is equivalent to the **x** command.

Another very useful operator is **c** or change. The command **cw** thus changes the text of a single word. You follow it by the replacement text ending with an ESC. Find a word which you can change to another, and try this now. Notice that the end of the text to be changed was marked with the character '$' so that you can see this as you are typing in the new material.

3.4. Operating on lines

It is often the case that you want to operate on lines. Find a line which you want to delete, and type **dd**, the **d** operator twice. This will delete the line. If you are on a dumb terminal, the editor may just erase the line on the screen, replacing it with a line with only an @ on it. This line does not correspond to any line in your file, but only acts as a place holder. It helps to avoid a lengthy redraw of the rest of the screen which would be necessary to close up the hole created by the deletion on a terminal without a delete line capability.

Try repeating the **c** operator twice; this will change a whole line, erasing its previous contents and replacing them with text you type up to an ESC.†

You can delete or change more than one line by preceding the **dd** or **cc** with a count, i.e. **5dd** deletes 5 lines. You can also give a command like **dL** to delete all the lines up to and including the last line on the screen, or **d3L** to delete through the third from the bottom line. Try some commands like this now.* Notice that the editor lets you know when you change a large number of lines so that you can see the extent of the change. The editor will also always tell you when a change you make affects text which you cannot see.

3.5. Undoing

Now suppose that the last change which you made was incorrect; you could use the insert, delete and append commands to put the correct material back. However, since it is often the case that we regret a change or make a change incorrectly, the editor provides a **u** (undo) command to reverse the last change which you made. Try this a few times, and give it twice in a row to notice that an **u** also undoes a **u**.

The undo command lets you reverse only a single change. After you make a number of changes to a line, you may decide that you would rather have the original state of the line back. The **U** command restores the current line to the state before you started changing it.

You can recover text which you delete, even if undo will not bring it back; see the section on recovering lost text below.

3.6. Summary

SPACE	advance the cursor one position
^H	backspace the cursor
^W	erase a word during an insert
erase	your erase (usually ^H or #), erases a character during an insert
kill	your kill (usually @, ^X, or ^U), kills the insert on this line
.	repeats the changing command
O	opens and inputs new lines, above the current
U	undoes the changes you made to the current line
a	appends text after the cursor
c	changes the object you specify to the following text

† The command S is a convenient synonym for for **cc**, by analogy with **s**. Think of S as a substitute on lines, while **s** is a substitute on characters.

* One subtle point here involves using the **/** search after a **d**. This will normally delete characters from the current position to the point of the match. If what is desired is to delete whole lines including the two points, give the pattern as **/pat/+0**, a line address.

d	deletes the object you specify
i	inserts text before the cursor
o	opens and inputs new lines, below the current
u	undoes the last change

4. Moving about; rearranging and duplicating text

4.1. Low level character motions

Now move the cursor to a line where there is a punctuation or a bracketing character such as a parenthesis or a comma or period. Try the command **f***x* where *x* is this character. This command finds the next *x* character to the right of the cursor in the current line. Try then hitting a **;**, which finds the next instance of the same character. By using the **f** command and then a sequence of **;**'s you can often get to a particular place in a line much faster than with a sequence of word motions or SPACES. There is also a **F** command, which is like **f**, but searches backward. The **;** command repeats **F** also.

When you are operating on the text in a line it is often desirable to deal with the characters up to, but not including, the first instance of a character. Try **df***x* for some *x* now and notice that the *x* character is deleted. Undo this with **u** and then try **dt***x*; the **t** here stands for to, i.e. delete up to the next *x*, but not the *x*. The command **T** is the reverse of **t**.

When working with the text of a single line, an **↑** moves the cursor to the first non-white position on the line, and a **$** moves it to the end of the line. Thus **$a** will append new text at the end of the current line.

Your file may have tab (^I) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every 8 positions.* When the cursor is at a tab, it sits on the last of the several spaces which represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

On rare occasions, your file may have nonprinting characters in it. These characters are displayed in the same way they are represented in this document, that is with a two character code, the first character of which is '^'. On the screen non-printing characters resemble a '^' character adjacent to another, but spacing or backspacing over the character will reveal that the two characters are, like the spaces representing a tab character, a single character.

The editor sometimes discards control characters, depending on the character and the setting of the *beautify* option, if you attempt to insert them in your file. You can get a control character in the file by beginning an insert and then typing a ^V before the control character. The ^V quotes the following character, causing it to be inserted directly into the file.

4.2. Higher level text objects

In working with a document it is often advantageous to work in terms of sentences, paragraphs, and sections. The operations **(** and **)** move to the beginning of the previous and next sentences respectively. Thus the command **d)** will delete the rest of the current sentence; likewise **d(** will delete the previous sentence if you are at the beginning of the current sentence, or the current sentence up to where you are if you are not at the beginning of the current sentence.

A sentence is defined to end at a '.', '!' or '?' which is followed by either the end of a line, or by two spaces. Any number of closing ')', ']', '"' and ''' characters may appear after the '.', '!' or '?' before the spaces or end of line.

The operations **{** and **}** move over paragraphs and the operations **[[** and **]]** move over sections.†

* This is settable by a command of the form **:se ts=***x*CR, where *x* is 4 to set tabstops every four columns. This has effect on the screen representation within the editor.

† The **[[** and **]]** operations require the operation character to be doubled because they can move the cursor far

A paragraph begins after each empty line, and also at each of a set of paragraph macros, specified by the pairs of characters in the definition of the string valued option *paragraphs*. The default setting for this option defines the paragraph macros of the *−ms* and *−mm* macro packages, i.e. the '.IP', '.LP', '.PP' and '.QP', '.P' and '.LI' macros.‡ Each paragraph boundary is also a sentence boundary. The sentence and paragraph commands can be given counts to operate over groups of sentences and paragraphs.

Sections in the editor begin after each macro in the *sections* option, normally '.NH', '.SH', '.H' and '.HU', and each line with a formfeed ^L in the first column. Section boundaries are always line and paragraph boundaries also.

Try experimenting with the sentence and paragraph commands until you are sure how they work. If you have a large document, try looking through it using the section commands. The section commands interpret a preceding count as a different window size in which to redraw the screen at the new location, and this window size is the base size for newly drawn windows until another size is specified. This is very useful if you are on a slow terminal and are looking for a particular section. You can give the first section command a small count to then see each successive section heading in a small window.

4.3. Rearranging and duplicating text

The editor has a single unnamed buffer where the last deleted or changed away text is saved, and a set of named buffers **a**−**z** which you can use to save copies of text and to move text around in your file and between files.

The operator y yanks a copy of the object which follows into the unnamed buffer. If preceded by a buffer name, "*x*y, where *x* here is replaced by a letter **a**−**z**, it places the text in the named buffer. The text can then be put back in the file with the commands **p** and **P**; **p** puts the text after or below the cursor, while **P** puts the text before or above the cursor.

If the text which you yank forms a part of a line, or is an object such as a sentence which partially spans more than one line, then when you put the text back, it will be placed after the cursor (or before if you use **P**). If the yanked text forms whole lines, they will be put back as whole lines, without changing the current line. In this case, the put acts much like a **o** or **O** command.

Try the command **YP**. This makes a copy of the current line and leaves you on this copy, which is placed before the current line. The command **Y** is a convenient abbreviation for **yy**. The command **Yp** will also make a copy of the current line, and place it after the current line. You can give **Y** a count of lines to yank, and thus duplicate several lines; try **3YP**.

To move text within the buffer, you need to delete it in one place, and put it back in another. You can precede a delete operation by the name of a buffer in which the text is to be stored as in **"a5dd** deleting 5 lines into the named buffer *a*. You can then move the cursor to the eventual resting place of the these lines and do a **"ap** or **"aP** to put them back. In fact, you can switch and edit another file before you put the lines back, by giving a command of the form **:e** *name*CR where *name* is the name of the other file you want to edit. You will have to write back the contents of the current editor buffer (or discard them) if you have made changes before the editor will let you switch to the other file. An ordinary delete command saves the text in the unnamed buffer, so that an ordinary put can move it elsewhere. However, the unnamed buffer is lost when you change files, so to move text from one file to another you should use an unnamed buffer.

---

from where it currently is. While it is easy to get back with the command **''**, these commands would still be frustrating if they were easy to hit accidentally.

‡ You can easily change or extend this set of macros by assigning a different string to the *paragraphs* option in your EXINIT. See section 6.2 for details. The '.bp' directive is also considered to start a paragraph.

4.4. Summary.

↑	first non-white on line
$	end of line
)	forward sentence
}	forward paragraph
]]	forward section
(	backward sentence
{	backward paragraph
[[	backward section
f*x*	find *x* forward in line
p	put text back, after cursor or below current line
y	yank operator, for copies and moves
t*x*	up to *x* forward, for operators
F*x*	f backward in line
P	put text back, before cursor or above current line
T*x*	t backward in line

5. High level commands

5.1. Writing, quitting, editing new files

So far we have seen how to enter *vi* and to write out our file using either ZZ or :wCR. The first exits from the editor, (writing if changes were made), the second writes and stays in the editor.

If you have changed the editor's copy of the file but do not wish to save your changes, either because you messed up the file or decided that the changes are not an improvement to the file, then you can give the command :q!CR to quit from the editor without writing the changes. You can also reedit the same file (starting over) by giving the command :e!CR. These commands should be used only rarely, and with caution, as it is not possible to recover the changes you have made after you discard them in this manner.

You can edit a different file without leaving the editor by giving the command :e *name*CR. If you have not written out your file before you try to do this, then the editor will tell you this, and delay editing the other file. You can then give the command :wCR to save your work and then the :e *name*CR command again, or carefully give the command :e! *name*CR, which edits the other file discarding the changes you have made to the current file. To have the editor automatically save changes, include *set autowrite* in your EXINIT, and use :n instead of :e.

5.2. Escaping to a shell

You can get to a shell to execute a single command by giving a *vi* command of the form :!*cmd*CR. The system will run the single command *cmd* and when the command finishes, the editor will ask you to hit a RETURN to continue. When you have finished looking at the output on the screen, you should hit RETURN and the editor will clear the screen and redraw it. You can then continue editing. You can also give another : command when it asks you for a RETURN; in this case the screen will not be redrawn.

If you wish to execute more than one command in the shell, then you can give the command :shCR. This will give you a new shell, and when you finish with the shell, ending it by typing a ^D, the editor will clear the screen and continue.

On systems which support it, ^Z will suspend the editor and return to the (top level) shell. When the editor is resumed, the screen will be redrawn.

5.3. Marking and returning

The command `` returned to the previous place after a motion of the cursor by a command such as /, ? or G. You can also mark lines in the file with single letter tags and return to these marks later by naming the tags. Try marking the current line with the command **m***x*, where you should pick some letter for *x*, say 'a'. Then move the cursor to a different line (any way you like) and hit `a. The cursor will return to the place which you marked. Marks last only until you edit another file.

When using operators such as **d** and referring to marked lines, it is often desirable to delete whole lines rather than deleting to the exact position in the line marked by **m**. In this case you can use the form ´*x* rather than `*x*. Used without an operator, ´*x* will move to the first non-white character of the marked line; similarly ´´ moves to the first non-white character of the line containing the previous context mark ``.

5.4. Adjusting the screen

If the screen image is messed up because of a transmission error to your terminal, or because some program other than the editor wrote output to your terminal, you can hit a ^L, the ASCII form-feed character, to cause the screen to be refreshed.

On a dumb terminal, if there are @ lines in the middle of the screen as a result of line deletion, you may get rid of these lines by typing ^R to cause the editor to retype the screen, closing up these holes.

Finally, if you wish to place a certain line on the screen at the top middle or bottom of the screen, you can position the cursor to that line, and then give a **z** command. You should follow the **z** command with a RETURN if you want the line to appear at the top of the window, a . if you want it at the center, or a − if you want it at the bottom. (z., z-, and z+ are not available on all v2 editors.)

6. Special topics

6.1. Editing on slow terminals

When you are on a slow terminal, it is important to limit the amount of output which is generated to your screen so that you will not suffer long delays, waiting for the screen to be refreshed. We have already pointed out how the editor optimizes the updating of the screen during insertions on dumb terminals to limit the delays, and how the editor erases lines to @ when they are deleted on dumb terminals.

The use of the slow terminal insertion mode is controlled by the *slowopen* option. You can force the editor to use this mode even on faster terminals by giving the command :se **slow**CR. If your system is sluggish this helps lessen the amount of output coming to your terminal. You can disable this option by :se **noslow**CR.

The editor can simulate an intelligent terminal on a dumb one. Try giving the command :se **redraw**CR. This simulation generates a great deal of output and is generally tolerable only on lightly loaded systems and fast terminals. You can disable this by giving the command
:se **noredraw**CR.

The editor also makes editing more pleasant at low speed by starting editing in a small window, and letting the window expand as you edit. This works particularly well on intelligent terminals. The editor can expand the window easily when you insert in the middle of the screen on these terminals. If possible, try the editor on an intelligent terminal to see how this works.

You can control the size of the window which is redrawn each time the screen is cleared by giving window sizes as argument to the commands which cause large screen motions:

```
    :  /  ?  [[  ]]  `  ´
```

Thus if you are searching for a particular instance of a common string in a file you can precede the first search command by a small number, say 3, and the editor will draw three line windows around each instance of the string which it locates.

You can easily expand or contract the window, placing the current line as you choose, by giving a number on a z command, after the z and before the following RETURN, . or –. Thus the command z5. redraws the screen with the current line in the center of a five line window.†

If the editor is redrawing or otherwise updating large portions of the display, you can interrupt this updating by hitting a DEL or RUB as usual. If you do this you may partially confuse the editor about what is displayed on the screen. You can still edit the text on the screen if you wish; clear up the confusion by hitting a ^L; or move or search again, ignoring the current state of the display.

See section 7.8 on *open* mode for another way to use the *vi* command set on slow terminals.

6.2. Options, set, and editor startup files

The editor has a set of options, some of which have been mentioned above. The most useful options are given in the following table.

Name	Default	Description
autoindent	noai	Supply indentation automatically
autowrite	noaw	Automatic write before :n, :ta, ^↑, !
ignorecase	noic	Ignore case in searching
lisp	nolisp	({) } commands deal with S-expressions
list	nolist	Tabs print as ^I; end of lines marked with S
magic	nomagic	The characters . [and * are special in scans
number	nonu	Lines are displayed prefixed with line numbers
paragraphs	para=IPLPPPQPbpP LI	Macro names which start paragraphs
redraw	nore	Simulate a smart terminal on a dumb one
sections	sect=NHSHH HU	Macro names which start new sections
shiftwidth	sw=8	Shift distance for <, > and input ^D and ^T
showmatch	nosm	Show matching (or { as) or } is typed
slowopen	slow	Postpone display updates during inserts
term	dumb	The kind of terminal you are using.

The options are of three kinds: numeric options, string options, and toggle options. You can set numeric and string options by a statement of the form

    set *opt*=*val*

and toggle options can be set or unset by statements of one of the forms

    set *opt*
    set no*opt*

These statements can be placed in your EXINIT in your environment, or given while you are running *vi* by preceding them with a : and following them with a CR.

You can get a list of all options which you have changed by the command :setCR, or the value of a single option by the command :set *opt*?CR. A list of all possible options and their values is generated by :set allCR. Set can be abbreviated se. Multiple options can be placed on one line, e.g. :se ai aw nuCR.

Options set by the set command only last while you stay in the editor. It is common to want to have certain options set whenever you use the editor. This can be accomplished by creating a list of *ex* commands† which are to be run every time you start up *ex*, *edit*, or *vi*. A

† Note that the command 5z. has an entirely different effect, placing line 5 in the center of a new window.
† All commands which start with : are *ex* commands.

typical list includes a set command, and possibly a few **map** commands (on v3 editors). Since it is advisable to get these commands on one line, they can be separated with the | character, for example:

```
set ai aw terse|map @ dd|map # x
```

which sets the options *autoindent*, *autowrite*, *terse*, (the set command), makes @ delete a line, (the first **map**), and makes # delete a character, (the second **map**). (See section 6.9 for a description of the **map** command, which only works in version 3.) This string should be placed in the variable EXINIT in your environment. If you use *csh*, put this line in the file *.login* in your home directory:

```
setenv EXINIT 'set ai aw terse|map @ dd|map # x'
```

If you use the standard v7 shell, put these lines in the file *.profile* in your home directory:

```
EXINIT='set ai aw terse|map @ dd|map # x'
export EXINIT
```

On a version 6 system, the concept of environments is not present. In this case, put the line in the file *.exrc* in your home directory.

```
set ai aw terse|map @ dd|map # x
```

Of course, the particulars of the line would depend on which options you wanted to set.

6.3. Recovering lost lines

You might have a serious problem if you delete a number of lines and then regret that they were deleted. Despair not, the editor saves the last 9 deleted blocks of text in a set of numbered registers 1−9. You can get the *n*'th previous deleted text back in your file by the command "*n***p**. The " here says that a buffer name is to follow, *n* is the number of the buffer you wish to try (use the number 1 for now), and **p** is the put command, which puts text in the buffer after the cursor. If this doesn't bring back the text you wanted, hit **u** to undo this and then **.** (period) to repeat the put command. In general the **.** command will repeat the last change you made. As a special case, when the last command refers to a numbered text buffer, the **.** command increments the number of the buffer before repeating the command. Thus a sequence of the form

```
"1pu.u.u.
```

will, if repeated long enough, show you all the deleted text which has been saved for you. You can omit the **u** commands here to gather up all this text in the buffer, or stop after any **.** command to keep just the then recovered text. The command **P** can also be used rather than **p** to put the recovered text before rather than after the cursor.

6.4. Recovering lost files

If the system crashes, you can recover the work you were doing to within a few changes. You will normally receive mail when you next login giving you the name of the file which has been saved for you. You should then change to the directory where you were when the system crashed and give a command of the form:

```
% vi −r name
```

replacing *name* with the name of the file which you were editing. This will recover your work to a point near where you left off.†

† In rare cases, some of the lines of the file may be lost. The editor will give you the numbers of these lines and the text of the lines will be replaced by the string 'LOST'. These lines will almost always be among the last few which you changed. You can either choose to discard the changes which you made (if they are easy to remake) or to replace the few lost lines by hand.

You can get a listing of the files which are saved for you by giving the command:

```
% vi −r
```

If there is more than one instance of a particular file saved, the editor gives you the newest instance each time you recover it. You can thus get an older saved copy back by first recovering the newer copies.

For this feature to work, *vi* must be correctly installed by a super user on your system, and the *mail* program must exist to receive mail. The invocation "*vi -r*" will not always list all saved files, but they can be recovered even if they are not listed.

6.5. Continuous text input

When you are typing in large amounts of text it is convenient to have lines broken near the right margin automatically. You can cause this to happen by giving the command :se **wm=10**CR. This causes all lines to be broken at a space at least 10 columns from the right hand edge of the screen.*

If the editor breaks an input line and you wish to put it back together you can tell it to join the lines with **J**. You can give **J** a count of the number of lines to be joined as in **3J** to join 3 lines. The editor supplies white space, if appropriate, at the juncture of the joined lines, and leaves the cursor at this white space. You can kill the white space with **x** if you don't want it.

6.6. Features for editing programs

The editor has a number of commands for editing programs. The thing that most distinguishes editing of programs from editing of text is the desirability of maintaining an indented structure to the body of the program. The editor has a *autoindent* facility for helping you generate correctly indented programs.

To enable this facility you can give the command :se aiCR. Now try opening a new line with **o** and type some characters on the line after a few tabs. If you now start another line, notice that the editor supplies white space at the beginning of the line to line it up with the previous line. You cannot backspace over this indentation, but you can use **^D** key to backtab over the supplied indentation.

Each time you type **^D** you back up one position, normally to an 8 column boundary. This amount is settable; the editor has an option called *shiftwidth* which you can set to change this value. Try giving the command :se sw=4CR and then experimenting with autoindent again.

For shifting lines in the program left and right, there are operators **<** and **>**. These shift the lines you specify right or left by one *shiftwidth*. Try **<<** and **>>** which shift one line left or right, and **<L** and **>L** shifting the rest of the display left and right.

If you have a complicated expression and wish to see how the parentheses match, put the cursor at a left or right parenthesis and hit **%**. This will show you the matching parenthesis. This works also for braces **{** and **}**, and brackets **[** and **]**.

If you are editing C programs, you can use the **[[** and **]]** keys to advance or retreat to a line starting with a **{**, i.e. a function declaration at a time. When **]]** is used with an operator it stops after a line which starts with **}**; this is sometimes useful with **y]]**.

* This feature is not available on some v2 editors. In v2 editors where it is available, the break can only occur to the right of the specified boundary instead of to the left.

6.7. Filtering portions of the buffer

You can run system commands over portions of the buffer using the operator **!**. You can use this to sort lines in the buffer, or to reformat portions of the buffer with a pretty-printer. Try typing in a list of random words, one per line and ending them with a blank line. Back up to the beginning of the list, and then give the command **!}sort**CR. This says to sort the next paragraph of material, and the blank line ends a paragraph.

6.8. Commands for editing LISP†

If you are editing a LISP program you should set the option *lisp* by doing **:se lisp**CR. This changes the **(** and **)** commands to move backward and forward over s-expressions. The **{** and **}** commands are like **(** and **)** but don't stop at atoms. These can be used to skip to the next list, or through a comment quickly.

The *autoindent* option works differently for LISP, supplying indent to align at the first argument to the last open list. If there is no such argument then the indent is two spaces more than the last level.

There is another option which is useful for typing in LISP, the *showmatch* option. Try setting it with **:se sm**CR and then try typing a '(' some words and then a ')'. Notice that the cursor shows the position of the '(' which matches the ')' briefly. This happens only if the matching '(' is on the screen, and the cursor stays there for at most one second.

The editor also has an operator to realign existing lines as though they had been typed in with *lisp* and *autoindent* set. This is the **=** operator. Try the command **=%** at the beginning of a function. This will realign all the lines of the function declaration.

When you are editing LISP,, the **[[** and **]]** advance and retreat to lines beginning with a **(**, and are useful for dealing with entire function definitions.

6.9. Macros‡

Vi has a parameterless macro facility, which lets you set it up so that when you hit a single keystroke, the editor will act as though you had hit some longer sequence of keys. You can set this up if you find yourself typing the same sequence of commands repeatedly.

Briefly, there are two flavors of macros:

a) Ones where you put the macro body in a buffer register, say *x*. You can then type **@x** to invoke the macro. The **@** may be followed by another **@** to repeat the last macro.

b) You can use the *map* command from *vi* (typically in your *EXINIT*) with a command of the form:

:map *lhs rhs*CR

mapping *lhs* into *rhs*. There are restrictions: *lhs* should be one keystroke (either 1 character or one function key) since it must be entered within one second (unless *notimeout* is set, in which case you can type it as slowly as you wish, and *vi* will wait for you to finish it before it echoes anything). The *lhs* can be no longer than 10 characters, the *rhs* no longer than 100. To get a space, tab or newline into *lhs* or *rhs* you should escape them with a ^V. (It may be necessary to double the ^V if the map command is given inside *vi*, rather than in *ex*.) Spaces and tabs inside the *rhs* need not be escaped.

Thus to make the **q** key write and exit the editor, you can give the command

:map q :wq^V^VCR CR

which means that whenever you type **q**, it will be as though you had typed the four characters **:wq**CR. A ^V's is needed because without it the CR would end the : command, rather than

† The LISP features are not available on some v2 editors due to memory constraints.

‡ The macro feature is available only in version 3 editors.

becoming part of the *map* definition. There are two ^V's because from within *vi*, two ^V's must be typed to get one. The first CR is part of the *rhs*, the second terminates the : command.

Macros can be deleted with

```
unmap lhs
```

If the *lhs* of a macro is "#0" through "#9", this maps the particular function key instead of the 2 character "#" sequence. So that terminals without function keys can access such definitions, the form "#x" will mean function key *x* on all terminals (and need not be typed within one second.) The character "#" can be changed by using a macro in the usual way:

```
:map ^V^V^I #
```

to use tab, for example. (This won't affect the *map* command, which still uses #, but just the invocation from visual mode.

The undo command reverses an entire macro call as a unit, if it made any changes.

Placing a '!' after the word **map** causes the mapping to apply to input mode, rather than command mode. Thus, to arrange for ^T to be the same as 4 spaces in input mode, you can type:

```
:map ^T ^Vƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀƀ
```

where ƀ is a blank. The ^V is necessary to prevent the blanks from being taken as white space between the *lhs* and *rhs*.

7. Word Abbreviations ‡‡

A feature similar to macros in input mode is word abbreviation. This allows you to type a short word and have it expanded into a longer word or words. The commands are **:abbreviate** and **:unabbreviate** (**:ab** and **:una**) and have the same syntax as **:map**. For example:

```
:ab eecs Electrical Engineering and Computer Sciences
```

causes the word 'eecs' to always be changed into the phrase 'Electrical Engineering and Computer Sciences'. Word abbreviation is different from macros in that only whole words are affected. If 'eecs' were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. There is no need for an abbreviation to be a single keystroke, as it should be with a macro.

7.1. Abbreviations

The editor has a number of short commands which abbreviate longer commands which we have introduced here. You can find these commands easily on the quick reference card. They often save a bit of typing and you can learn them as convenient.

8. Nitty-gritty details

8.1. Line representation in the display

The editor folds long logical lines onto many physical lines in the display. Commands which advance lines advance logical lines and will skip over all the segments of a line in one motion. The command | moves the cursor to a specific column, and may be useful for getting near the middle of a long line to split it in half. Try 80| on a line which is more than 80 columns long.†

The editor only puts full lines on the display; if there is not enough room on the display to fit a logical line, the editor leaves the physical line empty, placing only an @ on the line as a

‡‡ Version 3 only.

† You can make long lines very easily by using J to join together short lines.

place holder. When you delete lines on a dumb terminal, the editor will often just clear the lines to @ to save time (rather than rewriting the rest of the screen.) You can always maximize the information on the screen by giving the ^R command.

If you wish, you can have the editor place line numbers before each line on the display. Give the command :se **nu**CR to enable this, and the command :se **nonu**CR to turn it off. You can have tabs represented as ^I and the ends of lines indicated with '$' by giving the command :se **list**CR; :se **nolist**CR turns this off.

Finally, lines consisting of only the character '~' are displayed when the last line in the file is in the middle of the screen. These represent physical lines which are past the logical end of file.

8.2. Counts

Most *vi* commands will use a preceding count to affect their behavior in some way. The following table gives the common ways in which the counts are used:

new window size	: / ? [[ ]] \` '
scroll amount	^D ^U
line/column number	z G |
repeat effect	most of the rest

The editor maintains a notion of the current default window size. On terminals which run at speeds greater than 1200 baud the editor uses the full terminal screen. On terminals which are slower than 1200 baud (most dialup lines are in this group) the editor uses 8 lines as the default window size. At 1200 baud the default is 16 lines.

This size is the size used when the editor clears and refills the screen after a search or other motion moves far from the edge of the current window. The commands which take a new window size as count all often cause the screen to be redrawn. If you anticipate this, but do not need as large a window as you are currently using, you may wish to change the screen size by specifying the new size before these commands. In any case, the number of lines used on the screen will expand if you move off the top with a − or similar command or off the bottom with a command such as RETURN or ^D. The window will revert to the last specified size the next time it is cleared and refilled.†

The scroll commands ^D and ^U likewise remember the amount of scroll last specified, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus **10a+−−−−**ESC will insert a grid-like string of text. A few commands also use a preceding count as a line or column number.

Except for a few commands which ignore any counts (such as ^R), the rest of the editor commands use a count to indicate a simple repetition of their effect. Thus **5w** advances five words on the current line, while **5**RETURN advances five lines. A very useful instance of a count as a repetition is a count given to the **.** command, which repeats the last changing command. If you do **dw** and then **3.**, you will delete first one and then three words. You can then delete two more words with **2.**.

8.3. More file manipulation commands

The following table lists the file manipulation commands which you can use when you are in *vi*. All of these commands are followed by a CR or ESC. The most basic commands are :w and :e. A normal editing session on a single file will end with a **ZZ** command. If you are editing for a long period of time you can give :w commands occasionally after major amounts of editing, and then finish with a **ZZ**. When you edit more than one file, you can finish with one

† But not by a ^L which just redraws the screen as it is.

:w	write back changes
:wq	write and quit
:x	write (if necessary) and quit (same as ZZ).
:e *name*	edit file *name*
:e!	reedit, discarding changes
:e + *name*	edit, starting at end
:e +*n*	edit, starting at line *n*
:e #	edit alternate file
:w *name*	write file *name*
:w! *name*	overwrite file *name*
:*x,y*w *name*	write lines *x* through *y* to *name*
:r *name*	read file *name* into buffer
:r !*cmd*	read output of *cmd* into buffer
:n	edit next file in argument list
:n!	edit next file, discarding changes to current
:n *args*	specify new argument list
:ta *tag*	edit file containing tag *tag*, at *tag*

with a :w and start editing a new file by giving a :e command, or set *autowrite* and use :n <file>.

If you make changes to the editor's copy of a file, but do not wish to write them back, then you must give an ! after the command you would otherwise use; this forces the editor to discard any changes you have made. Use this carefully.

The :e command can be given a + argument to start at the end of the file, or a +*n* argument to start at line *n*. In actuality, *n* may be any editor command not containing a space, usefully a scan like +/*pat* or +?*pat*. In forming new names to the e command, you can use the character % which is replaced by the current file name, or the character # which is replaced by the alternate file name. The alternate file name is generally the last name you typed other than the current file. Thus if you try to do a :e and get a diagnostic that you haven't written the file, you can give a :w command and then a :e # command to redo the previous :e.

You can write part of the buffer to a file by finding out the lines that bound the range to be written using ^G, and giving these numbers after the : and before the w, separated by ,'s. You can also mark these lines with m and then use an address of the form '*x*,'*y* on the w command here.

You can read another file into the buffer after the current line by using the :r command. You can similarly read in the output from a command, just use !*cmd* instead of a file name.

If you wish to edit a set of files in succession, you can give all the names on the command line, and then edit each one in turn using the command :n. It is also possible to respecify the list of files to be edited by giving the :n command a list of file names, or a pattern to be expanded as you would have given it on the initial *vi* command.

If you are editing large programs, you will find the :ta command very useful. It utilizes a data base of function names and their locations, which can be created by programs such as *ctags*, to quickly find a function whose name you give. If the :ta command will require the editor to switch files, then you must :w or abandon any changes before switching. You can repeat the :ta command without any arguments to look for the same tag again. (The tag feature is not available in some v2 editors.)

8.4. More about searching for strings

When you are searching for strings in the file with / and ?, the editor normally places you at the next or previous occurrence of the string. If you are using an operator such as d, c or y, then you may well wish to affect lines up to the line before the line containing the pattern.

You can give a search of the form */pat/−n* to refer to the *n*'th line before the next line containing *pat*, or you can use + instead of − to refer to the lines after the one containing *pat*. If you don't give a line offset, then the editor will affect characters up to the match place, rather than whole lines; thus use "+0" to affect to the line which matches.

You can have the editor ignore the case of words in the searches it does by giving the command **:se ic**CR. The command **:se noic**CR turns this off.

Strings given to searches may actually be regular expressions. If you do not want or need this facility, you should

    set nomagic

in your EXINIT. In this case, only the characters ↑ and $ are special in patterns. The character \ is also then special (as it is most everywhere in the system), and may be used to get at the an extended pattern matching facility. It is also necessary to use a \ before a / in a forward scan or a ? in a backward scan, in any case. The following table gives the extended forms when **magic** is set.

↑	at beginning of pattern, matches beginning of line
$	at end of pattern, matches end of line
.	matches any character
\<	matches the beginning of a word
\>	matches the end of a word
[*str*]	matches any single character in *str*
[↑*str*]	matches any single character not in *str*
[*x*−*y*]	matches any character between *x* and *y*
*	matches any number of the preceding pattern

If you use **nomagic** mode, then the . [and * primitives are given with a preceding \.

8.5. More about input mode

There are a number of characters which you can use to make corrections during input mode. These are summarized in the following table.

^H	deletes the last input character
^W	deletes the last input word, defined as by **b**
erase	your erase character, same as ^H
kill	your kill character, deletes the input on this line
\	escapes a following ^H and your erase and kill
ESC	ends an insertion
DEL	interrupts an insertion, terminating it abnormally
CR	starts a new line
^D	backtabs over *autoindent*
0^D	kills all the *autoindent*
↑^D	same as 0^D, but restores indent next line
^V	quotes the next non-printing character into the file

The most usual way of making corrections to input is by typing ^H to correct a single character, or by typing one or more ^W's to back over incorrect words. If you use # as your erase character in the normal system, it will work like ^H.

Your system kill character, normally @, ^X or ^U, will erase all the input you have given on the current line. In general, you can neither erase input back around a line boundary nor can you erase characters which you did not insert with this insertion command. To make corrections on the previous line after a new line has been started you can hit ESC to end the insertion, move over and make the correction, and then return to where you were to continue.

The command A which appends at the end of the current line is often useful for continuing.

If you wish to type in your erase or kill character (say # or @) then you must precede it with a \, just as you would do at the normal system command level. A more general way of typing non-printing characters into the file is to precede them with a ^V. The ^V echoes as a ↑ character on which the cursor rests. This indicates that the editor expects you to type a control character. In fact you may type any character and it will be inserted into the file at that point.*

If you are using *autoindent* you can backtab over the indent which it supplies by typing a ^D. This backs up to a *shiftwidth* boundary. This only works immediately after the supplied *autoindent*.

When you are using *autoindent* you may wish to place a label at the left margin of a line. The way to do this easily is to type ↑ and then ^D. The editor will move the cursor to the left margin for one line, and restore the previous indent on the next. You can also type a 0 followed immediately by a ^D if you wish to kill all the indent and not have it come back on the next line.

8.6. Upper case only terminals

If your terminal has only upper case, you can still use *vi* by using the normal system convention for typing on such a terminal. Characters which you normally type are converted to lower case, and you can type upper case letters by preceding them with a \. The characters { ~ } | ` are not available on such terminals, but you can escape them as \(\↑ \) \! \'. These characters are represented on the display in the same way they are typed.‡ ‡

8.7. Vi and ex

Vi is actually one mode of editing within the editor *ex*. When you are running *vi* you can escape to the line oriented editor of *ex* by giving the command Q. All of the : commands which were introduced above are available in *ex*. Likewise, most *ex* commands can be invoked from *vi* using :. Just give them without the : and follow them with a CR.

In rare instances, an internal error may occur in *vi*. In this case you will get a diagnostic and be left in the command mode of *ex*. You can then save your work and quit if you wish by giving a command x after the : which *ex* prompts you with, or you can reenter *vi* by giving *ex* a *vi* command.

There are a number of things which you can do more easily in *ex* than in *vi*. Systematic changes in line oriented material are particularly easy. You can read the advanced editing documents for the editor *ed* to find out a lot more about this style of editing. Experienced users often mix their use of *ex* command mode and *vi* command mode to speed the work they are doing.

8.8. Open mode: vi on hardcopy terminals and "glass tty's" ‡

If you are on a hardcopy terminal or a terminal which does not have a cursor which can move off the bottom line, you can still use the command set of *vi*, but in a different mode. When you give a *vi* command, the editor will tell you that it is using *open* mode. This name comes from the *open* command in *ex*, which is used to get into the same mode.

The only difference between *visual* mode and *open* mode is the way in which the text is

* This is not quite true. The implementation of the editor does not allow the NULL (^@) character to appear in files. Also the LF (linefeed or ^J) character is used by the editor to separate lines in the file, so it cannot appear in the middle of a line. You can insert any other character, however, if you wait for the editor to echo the ↑ before you type the character. In fact, the editor will treat a following letter as a request for the corresponding control character. This is the only way to type ^S or ^Q, since the system normally uses them to suspend and resume output and never gives them to the editor to process.

‡ The \ character you give will not echo until you type another key.

‡ Not available in all v2 editors due to memory constraints.

displayed.

In *open* mode the editor uses a single line window into the file, and moving backward and forward in the file causes new lines to be displayed, always below the current line. Two commands of *vi* work differently in *open:* z and ^R. The z command does not take parameters, but rather draws a window of context around the current line and then returns you to the current line.

If you are on a hardcopy terminal, the ^R command will retype the current line. On such terminals, the editor normally uses two lines to represent the current line. The first line is a copy of the line as you started to edit it, and you work on the line below this line. When you delete characters, the editor types a number of \'s to show you the characters which are deleted. The editor also reprints the current line soon after such changes so that you can see what the line looks like again.

It is sometimes useful to use this mode on very slow terminals which can support *vi* in the full screen mode. You can do this by entering *ex* and using an *open* command.

Acknowledgements

Bruce Englar encouraged the early development of this display editor. Peter Kessler helped bring sanity to version 2's command layout. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and Unix systems.

Appendix: character functions

This appendix gives the uses the editor makes of each character. The characters are presented in their order in the ASCII character set: Control characters come first, then most special characters, then the digits, upper and then lower case characters.

For each character we tell a meaning it has as a command and any meaning it has during an insert. If it has only meaning as a command, then only this is discussed. Section numbers in parentheses indicate where the character is discussed; a 'f' after the section number means that the character is mentioned in a footnote.

^@	Not a command character. If typed as the first character of an insertion it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted the mechanism is not available. A ^@ cannot be part of the file due to the editor implementation (7.5f).
^A	Unused.
^B	Backward window. A count specifies repetition. Two lines of continuity are kept if possible (2.1, 6.1, 7.2).
^C	Unused.
^D	As a command, scrolls down a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future ^D and ^U commands (2.1, 7.2). During an insert, backtabs over *autoindent* white space at the beginning of a line (6.6, 7.5); this white space cannot be backspaced over.
^E	Exposes one more line below the current screen in the file, leaving the cursor where it is if possible. (Version 3 only.)
^F	Forward window. A count specifies repetition. Two lines of continuity are kept if possible (2.1, 6.1, 7.2).
^G	Equivalent to :fCR, printing the current file, whether it has been modified, the current line number and the number of lines in the file, and the percentage of the way through the file that you are.
^H (BS)	Same as **left arrow**. (See **h**). During an insert, eliminates the last input character, backing over it but not erasing it; it remains so you can see what you typed if you wish to type something only slightly different (3.1, 7.5).
^I (TAB)	Not a command character. When inserted it prints as some number of spaces. When the cursor is at a tab character it rests at the last of the spaces which represent the tab. The spacing of tabstops is controlled by the *tabstop* option (4.1, 6.6).
^J (LF)	Same as **down arrow** (see **j**).
^K	Unused.
^L	The ASCII formfeed character, this causes the screen to be cleared and redrawn. This is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt (5.4, 7.2f).
^M (CR)	A carriage return advances to the next line, at the first non-white position in the line. Given a count, it advances that many lines (2.3). During an insert, a CR causes the insert to continue onto another line (3.1).
^N	Same as **down arrow** (see **j**).
^O	Unused.

^P	Same as **up arrow** (see **k**).
^Q	Not a command character. In input mode, ^Q quotes the next character, the same as ^V, except that some teletype drivers will eat the ^Q so that the editor never sees it.
^R	Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line (5.4, 7.2, 7.8).
^S	Unused. Some teletype drivers use ^S to suspend output until ^Qis
^T	Not a command character. During an insert, with *autoindent* set and at the beginning of the line, inserts *shiftwidth* white space.
^U	Scrolls the screen up, inverting ^D which scrolls down. Counts work as they do for ^D, and the previous scroll amount is common to both. On a dumb terminal, ^U will often necessitate clearing and redrawing the screen further back in the file (2.1, 7.2).
^V	Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file (4.2, 7.5).
^W	Not a command character. During an insert, backs up as **b** would in command mode; the deleted characters remain on the display (see ^H) (7.5).
^X	Unused.
^Y	Exposes one more line above the current screen, leaving the cursor where it is if possible. (No mnemonic value for this key; however, it is next to ^U which scrolls up a bunch.) (Version 3 only.)
^Z	If supported by the Unix system, stops the editor, exiting to the top level shell. Same as **:stop**CR. Otherwise, unused.
^[(ESC)	Cancels a partially formed command, such as a **z** when no following character has yet been given; terminates inputs on the last line (read by commands such as **:** **/** and **?**); ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor rings the bell or flashes the screen. You can thus hit ESC if you don't know what is happening till the editor rings the bell. If you don't know if you are in insert mode you can type ESC**a**, and then material to be input; the material will be inserted correctly whether or not you were in insert mode when you started (1.5, 3.1, 7.5).
^\	Unused.
^]	Searches for the word which is after the cursor as a tag. Equivalent to typing **:ta**, this word, and then a CR. Mnemonically, this command is "go right to" (7.3).
^↑	Equivalent to **:e #**CR, returning to the previous position in the last edited file, or editing a file which you specified if you got a 'No write since last change diagnostic' and do not want to have to type the file name again (7.3). (You have to do a **:w** before ^↑ will work in this case. If you do not wish to write the file you should do **:e! #**CR instead.)
^_	Unused. Reserved as the command character for the Tektronix 4025 and 4027 terminal.
SPACE	Same as **right arrow** (see **l**).
!	An operator, which processes lines from the buffer with reformatting commands. Follow **!** with the object to be processed, and then the command name terminated by CR. Doubling **!** and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the **!**. Thus **2!}***fmt*CR reformats the next two paragraphs by running them through the program *fmt*. If you are working on LISP, the command **!%***grind*CR,* given at the

*Both *fmt* and *grind* are Berkeley programs and may not be present at all installations.

beginning of a function, will run the text of the function through the LISP grinder (6.7, 7.3). To read a file or the output of a command into the buffer use :r (7.3). To simply execute a command use :! (7.3).

"	Precedes a named buffer specification. There are named buffers **1**–**9** used for saving deleted text and named buffers **a**–**z** into which you can place text (4.3, 6.3)
#	The macro character which, when followed by a number, will substitute for a function key on terminals without function keys (6.9). In input mode, if this is your erase character, it will delete the last character you typed in input mode, and must be preceded with a \ to insert it, since it normally backs over the last input character you gave.
$	Moves to the end of the current line. If you **:se list**CR, then the end of each line will be shown by printing a **$** after the end of the displayed text in the line. Given a count, advances to the count'th following end of line; thus **2$** advances to the end of the following line.
%	Moves to the parenthesis or brace { } which balances the parenthesis or brace at the current cursor position.
&	A synonym for **:&**CR, by analogy with the *ex* **&** command.
'	When followed by a **'** returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a**–**z**, returns to the line which was marked with this letter with a **m** command, at the first non-white character in the line. (2.2, 5.3). When used with an operator such as **d**, the operation takes place over complete lines; if you use **\`**, the operation takes place from the exact marked place to the current cursor position within the line.
(	Retreats to the beginning of a sentence, or to the beginning of a LISP s-expression if the *lisp* option is set. A sentence ends at a **. ! ** or **?** which is followed by either the end of a line or by two spaces. Any number of closing **)] "** and **'** characters may appear after the **. !** or **?**, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see **{** and **[[** below). A count advances that many sentences (4.2, 6.8).
)	Advances to the beginning of a sentence. A count repeats the effect. See **(** above for the definition of a sentence (4.2, 6.8).
*	Unused.
+	Same as CR when used as a command.
,	Reverse of the last **f F t** or **T** command, looking the other way in the current line. Especially useful after hitting too many **;** characters. A count repeats the search.
–	Retreats to the previous line at the first non-white character. This is the inverse of **+** and RETURN. If the line moved to is not on the screen, the screen is scrolled, or cleared and redrawn if this is not possible. If a large amount of scrolling would be required the screen is also cleared and redrawn, with the current line at the center (2.3).
.	Repeats the last command which changed the buffer. Especially useful when deleting words or lines; you can delete some words/lines and then hit **.** to delete more and more words/lines. Given a count, it passes it on to the command being repeated. Thus after a **2dw**, **3.** deletes three words (3.3, 6.3, 7.2, 7.4).

/	Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The normal input editing sequences may be used during the input on the bottom line; an returns to command state without ever searching. The search begins when you hit CR to terminate the pattern; the cursor moves to the beginning of the last line to indicate that the search is in progress; the search may then be terminated with a DEL or RUB, or by backspacing when at the beginning of the bottom line, returning the cursor to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer. When used with an operator the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern you can force whole lines to be affected. To do this give a pattern with a closing a closing / and then an offset +*n* or −*n*. To include the character / in the search string, you must escape it with a preceding \\. A ↑ at the beginning of the pattern forces the match to occur at the beginning of a line only; this speeds the search. A $ at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available, see section 7.4; unless you set **nomagic** in your *.exrc* file you will have to preceed the characters . [* and ~ in the search pattern with a \\ to get them to work as you would naively expect (1.5, 2,2, 6.1, 7.2, 7.4).
0	Moves to the first character on the current line. Also used, in forming numbers, after an initial **1**−**9**.
1−9	Used to form numeric arguments to commands (2.3, 7.2).
:	A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with an CR, and the command then executed. You can return to where you were by hitting DEL or RUB if you hit **:** accidentally (see primarily 6.2 and 7.3).
;	Repeats the last single character find which used **f F t** or **T**. A count iterates the basic scan (4.1).
<	An operator which shifts lines left one *shiftwidth*, normally 8 spaces. Like all operators, affects lines when repeated, as in **<<**. Counts are passed through to the basic object, thus **3<<** shifts three lines (6.6, 7.2).
=	Reindents line for LISP, as though they were typed in with *lisp* and *autoindent* set (6.8).
>	An operator which shifts lines right one *shiftwidth*, normally 8 spaces. Affects lines when repeated as in **>>**. Counts repeat the basic object (6.6, 7.2).
?	Scans backwards, the opposite of **/**. See the **/** description above for details on scanning (2.2, 6.1, 7.4).
@	A macro character (6.9). If this is your kill character, you must escape it with a \\ to type it in during input mode, as it normally backs over the input you have given on the current line (3.1, 3.4, 7.5).
A	Appends at the end of line, a synonym for **$a** (7.2).
B	Backs up a word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the effect (2.4).
C	Changes the rest of the text on the current line; a synonym for **c$**.
D	Deletes the rest of the text on the current line; a synonym for **d$**.

E	Moves forward to the end of a word, defined as blanks and non-blanks, like B and W. A count repeats the effect.
F	Finds a single following character, backwards in the current line. A count repeats this search that many times (4.1).
G	Goes to the line number given as preceding argument, or the end of the file if no preceding count is given. The screen is redrawn with the new current line in the center if necessary (7.2).
H	**Home arrow.** Homes the cursor to the top line on the screen. If a count is given, then the cursor is moved to the count'th line on the screen. In any case the cursor is moved to the first non-white character on the line. If used as the target of an operator, full lines are affected (2.3, 3.2).
I	Inserts at the beginning of a line; a synonym for ↑**i**.
J	Joins together lines, supplying appropriate white space: one space between words, two spaces after a **.**, and no spaces at all if the first character of the joined on line is **)**. A count causes that many lines to be joined rather than the default two (6.5, 7.1f).
K	Unused.
L	Moves the cursor to the first non-white character of the last line on the screen. With a count, to the first non-white of the count'th line from the bottom. Operators affect whole lines when used with L (2.3).
M	Moves the cursor to the middle line on the screen, at the first non-white position on the line (2.3).
N	Scans for the next match of the last pattern given to **/** or **?**, but in the reverse direction; this is the reverse of **n**.
O	Opens a new line above the current line and inputs text there up to an ESC. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete, as the *slowopen* option works better (3.1).
P	Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise the text is inserted between the characters before and at the cursor. May be preceded by a named buffer specification "*x* to retrieve the contents of the buffer; buffers **1**−**9** contain deleted material, buffers **a**−**z** are available for general use (6.3).
Q	Quits from *vi* to *ex* command mode. In this mode, whole lines form commands, ending with a RETURN. You can give all the **:** commands; the editor supplies the **:** as a prompt (7.7).
R	Replaces characters on the screen with characters you type (overlay fashion). Terminates with an ESC.
S	Changes whole lines, a synonym for **cc**. A count substitutes for that many lines. The lines are saved in the numeric buffers, and erased on the screen before the substitution begins.
T	Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the effect. Most useful with operators such as **d** (4.1).
U	Restores the current line to its state before you started changing it (3.5).
V	Unused.

W	Moves forward to the beginning of a word in the current line, where words are defined as sequences of blank/non-blank characters. A count repeats the effect (2.4).
X	Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.
Y	Yanks a copy of the current line into the unnamed buffer, to be put back by a later **p** or **P**; a very useful synonym for **yy**. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer (7.4).
ZZ	Exits the editor. (Same as **:x**CR.) If any changes have been made, the buffer is written out to the current file. Then the editor quits.
[[	Backs up to the previous section boundary. A section begins at each macro in the *sections* option, normally a '.NH' or '.SH' and also at lines which which start with a formfeed ^L. Lines beginning with { also stop **[[**; this makes it useful for looking backwards, a function at a time, in C programs. If the option *lisp* is set, stops at each (at the beginning of a line, and is thus useful for moving backwards at the top level LISP objects. (4.2, 6.1, 6.6, 7.2).
\	Unused.
]]	Forward to a section boundary, see **[[** for a definition (4.2, 6.1, 6.6, 7.2).
↑	Moves to the first non-white position on the current line (4.4).
_	Unused.
`	When followed by a **`** returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a**–**z**, returns to the position which was marked with this letter with a **m** command. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line; if you use **'**, the operation takes place over complete lines (2.2, 5.3).
a	Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. The insertion terminates with an ESC (3.1, 7.2).
b	Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the effect (2.4).
c	An operator which changes the following object, replacing it with the following input text up to an ESC. If more than part of a single line is affected, the text which is changed away is saved in the numeric named buffers. If only part of the current line is affected, then the last character to be changed away is marked with a $. A count causes that many objects to be affected, thus both **3c)** and **c3)** change the following three sentences (7.4).
d	An operator which deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus **3dw** is the same as **d3w** (3.3, 3.4, 4.1, 7.4).
e	Advances to the end of the next word, defined as for **b** and **w**. A count repeats the effect (2.4, 3.1).
f	Finds the first instance of the next character following the cursor on the current line. A count repeats the find (4.1).
g	Unused.

Arrow keys **h**, **j**, **k**, **l**, and **H**.

h	**Left arrow**. Moves the cursor one character to the left. Like the other arrow keys, either **h**, the left arrow key, or one of the synonyms (^H) has the same effect. On v2 editors, arrow keys on certain kinds of terminals (those which send escape sequences, such as vt52, c100, or hp) cannot be used. A count repeats the effect (3.1, 7.5).
i	Inserts text before the cursor, otherwise like **a** (7.2).
j	**Down arrow**. Moves the cursor one line down in the same column. If the position does not exist, *vi* comes as close as possible to the same column. Synonyms include ^J (linefeed) and ^N.
k	**Up arrow**. Moves the cursor one line up. ^P is a synonym.
l	**Right arrow**. Moves the cursor one character to the right. SPACE is a synonym.
m	Marks the current position of the cursor in the mark register which is specified by the next character **a**–**z**. Return to this position or use with an operator using **`** or **'** (5.3).
n	Repeats the last **/** or **?** scanning commands (2.2).
o	Opens new lines below the current line; otherwise like **O** (3.1).
p	Puts text after/below the cursor; otherwise like **P** (6.3).
q	Unused.
r	Replaces the single character at the cursor with a single character you type. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see **R** above which is the more usually useful iteration of **r** (3.2).
s	Changes the single character under the cursor to the text which follows up to an ESC; given a count, that many characters from the current line are changed. The last character to be changed is marked with **$** as in **c** (3.2).
t	Advances the cursor upto the character before the next character typed. Most useful with operators such as **d** and **c** to delete the characters up to a following character. You can use **.** to delete more if this doesn't delete enough the first time (4.1).
u	Undoes the last change made to the current buffer. If repeated, will alternate between these two states, thus is its own inverse. When used after an insert which inserted text on more than one line, the lines are saved in the numeric named buffers (3.5).
v	Unused.
w	Advances to the beginning of the next word, as defined by **b** (2.4).
x	Deletes the single character under the cursor. With a count deletes deletes that many characters forward from the cursor position, but only on the current line (6.5).
y	An operator, yanks the following object into the unnamed temporary buffer. If preceded by a named buffer specification, "*x*, the text is placed in that buffer also. Text can be recovered by a later **p** or **P** (7.4).
z	Redraws the screen with the current line placed as specified by the following character: RETURN specifies the top of the screen, **.** the center of the screen, and **–** at the bottom of the screen. A count may be given after the **z** and before the following character to specify the new screen size for the redraw. A count before the **z** gives the number of the line to place in the center of the screen instead of the default current line. (5.4)

{	Retreats to the beginning of the beginning of the preceding paragraph. A paragraph begins at each macro in the *paragraphs* option, normally '.IP', '.LP', '.PP', '.QP' and '.bp'. A paragraph also begins after a completely empty line, and at each section boundary (see [[above) (4.2, 6.8, 7.6).
|	Places the cursor on the character in the column specified by the count (7.1, 7.2).
}	Advances to the beginning of the next paragraph. See { for the definition of paragraph (4.2, 6.8, 7.6).
~	Unused.
^? (DEL)	Interrupts the editor, returning it to command accepting state (1.5, 7.5)

Ex Reference Manual
Version 3.5/2.13 — September, 1980

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, Ca. 94720

ABSTRACT

Ex a line oriented text editor, which supports both command and display oriented editing. This reference manual describes the command oriented part of *ex;* the display editing features of *ex* are described in *An Introduction to Display Editing with Vi.* Other documents about the editor include the introduction *Edit: A tutorial*, the *Ex/edit Command Summary*, and a *Vi Quick Reference* card.

September 16, 1980

Ex Reference Manual
Version 3.5/2.13 — September, 1980

William Joy

Revised for versions 3.5/2.13 by
Mark Horton

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, Ca. 94720

1. Starting ex

Each instance of the editor has a set of options, which can be set to tailor it to your liking. The command *edit* invokes a version of *ex* designed for more casual or beginning users by changing the default settings of some of these options. To simplify the description which follows we assume the default settings of the options.

When invoked, *ex* determines the terminal type from the TERM variable in the environment. It there is a TERMCAP variable in the environment, and the type of the terminal described there matches the TERM variable, then that description is used. Also if the TERMCAP variable contains a pathname (beginning with a /) then the editor will seek the description of the terminal in that file (rather than the default /etc/termcap.) If there is a variable EXINIT in the environment, then the editor will execute the commands in that variable, otherwise if there is a file *.exrc* in your HOME directory *ex* reads commands from that file, simulating a *source* command. Option setting commands placed in EXINIT or *.exrc* will be executed before each editor session.

A command to enter *ex* has the following prototype:†

ex [**−**] [**−v**] [**−t** *tag*] [**−r**] [**−l**] [**−w***n*] [**−x**] [**−R**] [**+***command*] name ...

The most common case edits a single file with no options, i.e.:

ex name

The **−** command line option option suppresses all interactive-user feedback and is useful in processing editor scripts in command files. The **−v** option is equivalent to using *vi* rather than *ex.* The **−t** option is equivalent to an initial *tag* command, editing the file containing the *tag* and positioning the editor at its definition. The **−r** option is used in recovering after an editor or system crash, retrieving the last saved version of the named file or, if no file is specified, typing a list of saved files. The **−l** option sets up for editing LISP, setting the *showmatch* and *lisp* options. The **−w** option sets the default window size to *n,* and is useful on dialups to start in small windows. The **−x** option causes *ex* to prompt for a *key,* which is used to encrypt and decrypt the contents of the file, which should already be encrypted using the same key, see *crypt*(1). The **−R** option sets the *readonly* option at the start. ‡ *Name* arguments indicate files to be edited. An argument of the form **+***command* indicates that the editor should begin by

The financial support of an IBM Graduate Fellowship and the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 is gratefully acknowledged.

† Brackets '[' ']' surround optional parameters here.

‡ Not available in all v2 editors due to memory constraints.

executing the specified command. If *command* is omitted, then it defaults to "$", positioning the editor at the last line of the first file initially. Other useful commands here are scanning patterns of the form "/pat" or line numbers, e.g. "+100" starting at line 100.

2. File manipulation

2.1. Current file

Ex is normally editing the contents of a single file, whose name is recorded in the *current* file name. *Ex* performs all editing actions in a buffer (actually a temporary file) into which the text of the file is initially read. Changes made to the buffer have no effect on the file being edited unless and until the buffer contents are written out to the file with a *write* command. After the buffer contents are written, the previous contents of the written file are no longer accessible. When a file is edited, its name becomes the current file name, and its contents are read into the buffer.

The current file is almost always considered to be *edited.* This means that the contents of the buffer are logically connected with the current file name, so that writing the current buffer contents onto that file, even if it exists, is a reasonable action. If the current file is not *edited* then *ex* will not normally write on it if it already exists.*

2.2. Alternate file

Each time a new value is given to the current file name, the previous current file name is saved as the *alternate* file name. Similarly if a file is mentioned but does not become the current file, it is saved as the alternate file name.

2.3. Filename expansion

Filenames within the editor may be specified using the normal shell expansion conventions. In addition, the character '%' in filenames is replaced by the *current* file name and the character '#' by the *alternate* file name.†

2.4. Multiple files and named buffers

If more than one file is given on the command line, then the first file is edited as described above. The remaining arguments are placed with the first file in the *argument list.* The current argument list may be displayed with the *args* command. The next file in the argument list may be edited with the *next* command. The argument list may also be respecified by specifying a list of names to the *next* command. These names are expanded, the resulting list of names becomes the new argument list, and *ex* edits the first file on the list.

For saving blocks of text while editing, and especially when editing more than one file, *ex* has a group of named buffers. These are similar to the normal buffer, except that only a limited number of operations are available on them. The buffers have names *a* through *z.*‡

2.5. Read only

It is possible to use *ex* in *read only* mode to look at files that you have no intention of modifying. This mode protects you from accidently overwriting the file. Read only mode is on when the *readonly* option is set. It can be turned on with the **−R** command line option, by the *view* command line invocation, or by setting the *readonly* option. It can be cleared by setting *noreadonly.* It is possible to write, even while in read only mode, by indicating that you really

* The *file* command will say "[Not edited]" if the current file is not considered edited.

† This makes it easy to deal alternately with two files and eliminates the need for retyping the name supplied on an *edit* command after a *No write since last change* diagnostic is received.

‡ It is also possible to refer to *A* through *Z;* the upper case buffers are the same as the lower but commands append to named buffers rather than replacing if upper case names are used.

know what you are doing. You can write to a different file, or can use the ! form of write, even while in read only mode.

3. Exceptional Conditions

3.1. Errors and interrupts

When errors occur *ex* (optionally) rings the terminal bell and, in any case, prints an error diagnostic. If the primary input is from a file, editor processing will terminate. If an interrupt signal is received, *ex* prints "Interrupt" and returns to its command level. If the primary input is a file, then *ex* will exit when this occurs.

3.2. Recovering from hangups and crashes

If a hangup signal is received and the buffer has been modified since it was last written out, or if the system crashes, either the editor (in the first case) or the system (after it reboots in the second) will attempt to preserve the buffer. The next time you log in you should be able to recover the work you were doing, losing at most a few lines of changes from the last point before the hangup or editor crash. To recover a file you can use the **−r** option. If you were editing the file *resume,* then you should change to the directory where you were when the crash occurred, giving the command

ex −r *resume*

After checking that the retrieved file is indeed ok, you can *write* it over the previous contents of that file.

You will normally get mail from the system telling you when a file has been saved after a crash. The command

ex −r

will print a list of the files which have been saved for you. (In the case of a hangup, the file will not appear in the list, although it can be recovered.)

4. Editing modes

Ex has five distinct modes. The primary mode is *command* mode. Commands are entered in command mode when a ':' prompt is present, and are executed each time a complete line is sent. In *text input* mode *ex* gathers input lines and places them in the file. The *append, insert,* and *change* commands use text input mode. No prompt is printed when you are in text input mode. This mode is left by typing a '.' alone at the beginning of a line, and *command* mode resumes.

The last three modes are *open* and *visual* modes, entered by the commands of the same name, and, within open and visual modes *text insertion* mode. *Open* and *visual* modes allow local editing operations to be performed on the text in the file. The *open* command displays one line at a time on any terminal while *visual* works on CRT terminals with random positioning cursors, using the screen as a (single) window for file editing changes. These modes are described (only) in *An Introduction to Display Editing with Vi.*

5. Command structure

Most command names are English words, and initial prefixes of the words are acceptable abbreviations. The ambiguity of abbreviations is resolved in favor of the more commonly used commands.*

* As an example, the command *substitute* can be abbreviated 's' while the shortest available abbreviation for the *set* command is 'se'.

5.1. Command parameters

Most commands accept prefix addresses specifying the lines in the file upon which they are to have effect. The forms of these addresses will be discussed below. A number of commands also may take a trailing *count* specifying the number of lines to be involved in the command.† Thus the command "10p" will print the tenth line in the buffer while "delete 5" will delete five lines from the buffer, starting with the current line.

Some commands take other information or parameters, this information always being given after the command name.‡

5.2. Command variants

A number of commands have two distinct variants. The variant form of the command is invoked by placing an '!' immediately after the command name. Some of the default variants may be controlled by options; in this case, the '!' serves to toggle the default.

5.3. Flags after commands

The characters '#', 'p' and 'l' may be placed after many commands.** In this case, the command abbreviated by these characters is executed after the command completes. Since *ex* normally prints the new current line after each change, 'p' is rarely necessary. Any number of '+' or '−' characters may also be given with these flags. If they appear, the specified offset is applied to the current line value before the printing command is executed.

5.4. Comments

It is possible to give editor commands which are ignored. This is useful when making complex editor scripts for which comments are desired. The comment character is the double quote: ". Any command line beginning with " is ignored. Comments beginning with " may also be placed at the ends of commands, except in cases where they could be confused as part of text (shell escapes and the substitute and map commands).

5.5. Multiple commands per line

More than one command may be placed on a line by separating each pair of commands by a '|' character. However the *global* commands, comments, and the shell escape '!' must be the last command on a line, as they are not terminated by a '|'.

5.6. Reporting large changes

Most commands which change the contents of the editor buffer give feedback if the scope of the change exceeds a threshold given by the *report* option. This feedback helps to detect undesirably large changes so that they may be quickly and easily reversed with an *undo.* After commands with more global effect such as *global* or *visual,* you will be informed if the net change in the number of lines in the buffer during this command exceeds this threshold.

6. Command addressing

6.1. Addressing primitives

.	The current line. Most commands leave the current line as the last line which they affect. The default address for most commands is the current line, thus '.' is rarely used alone as an address.

† Counts are rounded down if necessary.
‡ Examples would be option names in a *set* command i.e. "set number", a file name in an *edit* command, a regular expression in a *substitute* command, or a target address for a *copy* command, i.e. "1,5 copy 25".
** A 'p' or 'l' must be preceded by a blank or tab except in the single special case 'dp'.

n	The *n*th line in the editor's buffer, lines being numbered sequentially from 1.
$	The last line in the buffer.
%	An abbreviation for "1,$", the entire buffer.
$+n$ $-n$	An offset relative to the current buffer line.†
/pat/ *?pat?*	Scan forward and backward respectively for a line containing *pat*, a regular expression (as defined below).. The scans normally wrap around the end of the buffer. If all that is desired is to print the next line containing *pat*, then the trailing / or ? may be omitted. If *pat* is omitted or explicitly empty, then the last regular expression specified is located.‡
'' 'x	**Before each non-relative motion of the current line '.', the previous current line is marked with a tag, subsequently referred to as ''''. This makes it easy to refer or return to this previous context. Marks may also be established by the *mark* command, using single lower case letters *x* and the marked lines referred to as '*'x*'.**

6.2. Combining addressing primitives

Addresses to commands consist of a series of addressing primitives, separated by ',' or ';'. Such address lists are evaluated left-to-right. When addresses are separated by ';' the current line '.' is set to the value of the previous addressing expression before the next address is interpreted. If more addresses are given than the command requires, then all but the last one or two are ignored. If the command takes two addresses, the first addressed line must precede the second in the buffer.†

7. Command descriptions

The following form is a prototype for all *ex* commands:

address **command** *! parameters count flags*

All parts are optional; the degenerate case is the empty command which prints the next line in the file. For sanity with use from within *visual* mode, *ex* ignores a ":" preceding any command.

In the following command descriptions, the default addresses are shown in parentheses, which are *not,* however, part of the command.

abbreviate *word rhs* abbr: **ab**

Add the named abbreviation to the current list. When in input mode in visual, if *word* is typed as a complete word, it will be changed to *rhs.*

(.) **append** abbr: **a**
text
.

Reads the input text and places it after the specified line. After the command, '.' addresses the last line input or the specified line if no lines were input. If address '0' is given, text is placed at the beginning of the buffer.

† The forms '.+3' '+3' and '+++' are all equivalent; if the current line is line 100 they all address line 103.
‡ The forms \/ and \? scan using the last regular expression used in a scan; after a substitute // and ?? would scan using the substitute's regular expression.
† Null address specifications are permitted in a list of addresses, the default in this case is the current line '.'; thus ',100' is equivalent to '.,100'. It is an error to give a prefix address to a command which expects none.

a!
text
.

The variant flag to *append* toggles the setting for the *autoindent* option during the input of *text.*

args

The members of the argument list are printed, with the current argument delimited by '[' and ']'.

(. , .) **change** *count* abbr: **c**
text
.

Replaces the specified lines with the input *text.* The current line becomes the last line input; if no lines were input it is left as for a *delete.*

c!
text
.

The variant toggles *autoindent* during the *change.*

(. , .) **copy** *addr flags* abbr: **co**

A *copy* of the specified lines is placed after *addr,* which may be '0'. The current line '.' addresses the last line of the copy. The command *t* is a synonym for *copy.*

(. , .) **delete** *buffer count flags* abbr: **d**

Removes the specified lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line. If a named *buffer* is specified by giving a letter, then the specified lines are saved in that buffer, or appended to it if an upper case letter is used.

edit *file* abbr: **e**
ex *file*

Used to begin an editing session on a new file. The editor first checks to see if the buffer has been modified since the last *write* command was issued. If it has been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file and prints the new filename. After insuring that this file is sensible† the editor reads the file into its buffer.

If the read of the file completes without error, the number of lines and characters read is typed. If there were any non-ASCII characters in the file they are stripped of their non-ASCII high bits, and any null characters in the file are discarded. If none of these errors occurred, the file is considered *edited.* If the last line of the input file is missing the trailing newline character, it will be supplied and a complaint will be issued. This command leaves the current line '.' at the last line read.‡

† I.e., that it is not a binary file such as a directory, a block or character special file other than */dev/tty,* a terminal, or a binary or executable file (as indicated by the first word).
‡ If executed from within *open* or *visual,* the current line is initially the first line of the file.

e! *file*

The variant form suppresses the complaint about modifications having been made and not written from the editor buffer, thus discarding all changes which have been made before editing the new file.

e +*n file*

Causes the editor to begin at line *n* rather than at the last line; *n* may also be an editor command containing no spaces, e.g.: "+/pat".

file abbr: **f**

Prints the current file name, whether it has been '[Modified]' since the last *write* command, whether it is *read only*, the current line, the number of lines in the buffer, and the percentage of the way through the buffer of the current line.*

file *file*

The current file name is changed to *file* which is considered '[Not edited]'.

(1 , $) **global** /*pat*/ *cmds* abbr: **g**

First marks each line among those specified which matches the given regular expression. Then the given command list is executed with '.' initially set to each marked line.

The command list consists of the remaining commands on the current input line and may continue to multiple lines by ending all but the last such line with a '\'. If *cmds* (and possibly the trailing / delimiter) is omitted, each line matching *pat* is printed. *Append, insert,* and *change* commands and associated input are permitted; the '.' terminating input may be omitted if it would be on the last line of the command list. *Open* and *visual* commands are permitted in the command list and take input from the terminal.

The *global* command itself may not appear in *cmds*. The *undo* command is also not permitted there, as *undo* instead can be used to reverse the entire *global* command. The options *autoprint* and *autoindent* are inhibited during a *global,* (and possibly the trailing / delimiter) and the value of the *report* option is temporarily infinite, in deference to a *report* for the entire global. Finally, the context mark '′′' is set to the value of '.' before the global command begins and is not changed during a global command, except perhaps by an *open* or *visual* within the *global.*

g! /*pat*/ *cmds* abbr: **v**

The variant form of *global* runs *cmds* at each line not matching *pat.*

(.) **insert** abbr: **i**
text
.

Places the given text before the specified line. The current line is left at the last line input; if there were none input it is left at the line before the addressed line. This command differs from *append* only in the placement of text.

* In the rare case that the current file is '[Not edited]' this is noted also; in this case you have to use the form w! to write to the file, since the editor is not sure that a **write** will not destroy a file unrelated to the current contents of the buffer.

i!
text
.

The variant toggles *autoindent* during the *insert*.

(. , .+1) **join** *count flags* abbr: **j**

Places the text from a specified range of lines together on one line. White space is adjusted at each junction to provide at least one blank character, two if there was a '.' at the end of the line, or none if the first following character is a ')'. If there is already white space at the end of the line, then the white space at the start of the next line will be discarded.

j!

The variant causes a simpler *join* with no white space processing; the characters in the lines are simply concatenated.

(.) **k** *x*

The *k* command is a synonym for *mark*. It does not require a blank or tab before the following letter.

(. , .) **list** *count flags*

Prints the specified lines in a more unambiguous way: tabs are printed as '^I' and the end of each line is marked with a trailing '$'. The current line is left at the last line printed.

map *lhs rhs*

The *map* command is used to define macros for use in *visual* mode. *Lhs* should be a single character, or the sequence "#n", for n a digit, referring to function key *n*. When this character or function key is typed in *visual* mode, it will be as though the corresponding *rhs* had been typed. On terminals without function keys, you can type "#n". See section 6.9 of the "Introduction to Display Editing with Vi" for more details.

(.) **mark** *x*

Gives the specified line mark *x*, a single lower case letter. The *x* must be preceded by a blank or a tab. The addressing form '´x' then addresses this line. The current line is not affected by this command.

(. , .) **move** *addr* abbr: **m**

The *move* command repositions the specified lines to be after *addr*. The first of the moved lines becomes the current line.

next abbr: **n**

The next file from the command line argument list is edited.

n!

The variant suppresses warnings about the modifications to the buffer not having been written out, discarding (irretrievably) any changes which may have been made.

n *filelist*
n +*command filelist*

The specified *filelist* is expanded and the resulting list replaces the current argument list; the first file in the new list is then edited. If *command* is given (it must contain no spaces), then it is executed after editing the first such file.

(. , .) **number** *count flags* abbr: **#** or **nu**

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

(.) **open** *flags* abbr: **o**
(.) **open** */pat/ flags*

Enters intraline editing *open* mode at each addressed line. If *pat* is given, then the cursor will be placed initially at the beginning of the string matched by the pattern. To exit this mode use Q. See *An Introduction to Display Editing with Vi* for more details. ‡

preserve

The current editor buffer is saved as though the system had just crashed. This command is for use only in emergencies when a *write* command has resulted in an error and you don't know how to save your work. After a *preserve* you should seek help.

(. , .) **print** *count* abbr: **p** or **P**

Prints the specified lines with non-printing characters printed as control characters '^*x*'; delete (octal 177) is represented as '^?'. The current line is left at the last line printed.

(.) **put** *buffer* abbr: **pu**

Puts back previously *deleted* or *yanked* lines. Normally used with *delete* to effect movement of lines, or with *yank* to effect duplication of lines. If no *buffer* is specified, then the last *deleted* or *yanked* text is restored.* By using a named buffer, text may be restored that was saved there at any previous time.

quit abbr: **q**

Causes *ex* to terminate. No automatic write of the editor buffer to a file is performed. However, *ex* issues a warning message if the file has changed since the last *write* command was issued, and does not *quit.*† Normally, you will wish to save your changes, and you should give a *write* command; if you wish to discard them, use the **q!** command variant.

q!

Quits from the editor, discarding changes to the buffer without complaint.

(.) **read** *file* abbr: **r**

Places a copy of the text of the given file in the editing buffer after the specified line. If no *file* is given the current file name is used. The current file name is not changed unless there is none in which case *file* becomes the current name. The sensibility restrictions for the *edit* command apply here also. If the file buffer is empty and there is no current name then *ex* treats this as an *edit* command.

‡ Not available in all v2 editors due to memory constraints.
* But no modifying commands may intervene between the *delete* or *yank* and the *put*, nor may lines be moved between files without using a named buffer.
† *Ex* will also issue a diagnostic if there are more files in the argument list.

Address '0' is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the *edit* command when the *read* successfully terminates. After a *read* the current line is the last line read.‡

(.) **read** !*command*

Reads the output of the command *command* into the buffer after the specified line. This is not a variant form of the command, rather a read specifying a *command* rather than a *filename;* a blank or tab before the ! is mandatory.

recover *file*

Recovers *file* from the system save area. Used after a accidental hangup of the phone** or a system crash** or *preserve* command. Except when you use *preserve* you will be notified by mail when a file is saved.

rewind abbr: **rew**

The argument list is rewound, and the first file in the list is edited.

rew!

Rewinds the argument list discarding any changes made to the current buffer.

set *parameter*

With no arguments, prints those options whose values have been changed from their defaults; with parameter *all* it prints all of the option values.

Giving an option name followed by a '?' causes the current value of that option to be printed. The '?' is unnecessary unless the option is Boolean valued. Boolean options are given values either by the form 'set *option*' to turn them on or 'set no*option*' to turn them off; string and numeric options are assigned via the form 'set *option*=value'.

More than one parameter may be given to *set*; they are interpreted left-to-right.

shell abbr: **sh**

A new shell is created. When it terminates, editing resumes.

source *file* abbr: **so**

Reads and executes commands from the specified file. *Source* commands may be nested.

(. , .) **substitute** */pat/repl/ options count flags* abbr: **s**

On each specified line, the first instance of pattern *pat* is replaced by replacement pattern *repl.* If the *global* indicator option character 'g' appears, then all instances are substituted; if the *confirm* indication character 'c' appears, then before each substitution the line to be substituted is typed with the string to be substituted marked with '↑' characters. By typing an 'y' one can cause the substitution to be performed, any other input causes no change to take place. After a *substitute* the current line is the last line substituted.

Lines may be split by substituting new-line characters into them. The newline in *repl* must be escaped by preceding it with a '\'. Other metacharacters available in *pat* and *repl* are described below.

‡ Within *open* and *visual* the current line is set to the first line read rather than the last.

** The system saves a copy of the file you were editing only if you have made changes to the file.

stop

Suspends the editor, returning control to the top level shell. If *autowrite* is set and there are unsaved changes, a write is done first unless the form **stop**! is used. This commands is only available where supported by the teletype driver and operating system.

(. , .) **substitute** *options count flags* abbr: **s**

If *pat* and *repl* are omitted, then the last substitution is repeated. This is a synonym for the **&** command.

(. , .) **t** *addr flags*

The *t* command is a synonym for *copy*.

ta *tag*

The focus of editing switches to the location of *tag*, switching to a different line in the current file where it is defined, or if necessary to another file.‡

The tags file is normally created by a program such as *ctags*, and consists of a number of lines with three fields separated by blanks or tabs. The first field gives the name of the tag, the second the name of the file where the tag resides, and the third gives an addressing form which can be used by the editor to find the tag; this field is usually a contextual scan using '*/pat/*' to be immune to minor changes in the file. Such scans are always performed as if *nomagic* was set.

The tag names in the tags file must be sorted alphabetically. ‡

unabbreviate *word* abbr: **una**

Delete *word* from the list of abbreviations.

undo abbr: **u**

Reverses the changes made in the buffer by the last buffer editing command. Note that *global* commands are considered a single command for the purpose of *undo* (as are *open* and *visual.)* Also, the commands *write* and *edit* which interact with the file system cannot be undone. *Undo* is its own inverse.

Undo always marks the previous value of the current line '.' as ''''. After an *undo* the current line is the first line restored or the line before the first line deleted if no lines were restored. For commands with more global effect such as *global* and *visual* the current line regains it's pre-command value after an *undo*.

unmap *lhs*

The macro expansion associated by *map* for *lhs* is removed.

(1 , $) **v** */pat/ cmds*

A synonym for the *global* command variant **g!**, running the specified *cmds* on each line which does not match *pat*.

version abbr: **ve**

Prints the current version number of the editor as well as the date the editor was last changed.

‡ If you have modified the current file before giving a *tag* command, you must write it out; giving another *tag* command, specifying no *tag* will reuse the previous tag.
‡ Not available in all v2 editors due to memory constraints.

(.) **visual** *type count flags* abbr: **vi**

Enters visual mode at the specified line. *Type* is optional and may be '−' , '↑' or '.' as in the *z* command to specify the placement of the specified line on the screen. By default, if *type* is omitted, the specified line is placed as the first on the screen. A *count* specifies an initial window size; the default is the value of the option *window*. See the document *An Introduction to Display Editing with Vi* for more details. To exit this mode, type Q.

visual file
visual +*n* file

From visual mode, this command is the same as edit.

(1 , $) **write** *file* abbr: **w**

Writes changes made back to *file*, printing the number of lines and characters written. Normally *file* is omitted and the text goes back where it came from. If a *file* is specified, then text will be written to that file.* If the file does not exist it is created. The current file name is changed only if there is no current file name; the current line is never changed.

If an error occurs while writing the current and *edited* file, the editor considers that there has been "No write since last change" even if the buffer had not previously been modified.

(1 , $) **write>>** *file* abbr: **w>>**

Writes the buffer contents at the end of an existing file.

w! *name*

Overrides the checking of the normal *write* command, and will write to any file which the system permits.

(1 , $) **w** **!***command*

Writes the specified lines into *command.* Note the difference between **w!** which overrides checks and **w** **!** which writes to a command.

wq *name*

Like a *write* and then a *quit* command.

wq! *name*

The variant overrides checking on the sensibility of the *write* command, as **w!** does.

xit *name*

If any changes have been made and not written, writes the buffer out. Then, in any case, quits.

(. , .) **yank** *buffer count* abbr: **ya**

Places the specified lines in the named *buffer,* for later retrieval via *put.* If no buffer name is specified, the lines go to a more volatile place; see the *put* command description.

* The editor writes to a file only if it is the current file and is *edited*, if the file does not exist, or if the file is actually a teletype, */dev/tty, /dev/null.* Otherwise, you must give the variant form **w!** to force the write.

(.+1) z *count*

Print the next *count* lines, default *window.*

(.) z *type count*

Prints a window of text with the specified line at the top. If *type* is '−' the line is placed at the bottom; a '.' causes the line to be placed in the center.* A count gives the number of lines to be displayed rather than double the number specified by the *scroll* option. On a CRT the screen is cleared before display begins unless a count which is less than the screen size is given. The current line is left at the last line printed.

! *command*

The remainder of the line after the '!' character is sent to a shell to be executed. Within the text of *command* the characters '%' and '#' are expanded as in filenames and the character '!' is replaced with the text of the previous command. Thus, in particular, '!!' repeats the last such shell escape. If any such expansion is performed, the expanded line will be echoed. The current line is unchanged by this command.

If there has been "[No write]" of the buffer contents since the last change to the editing buffer, then a diagnostic will be printed before the command is executed as a warning. A single '!' is printed when the command completes.

(*addr* , *addr*) ! *command*

Takes the specified address range and supplies it as standard input to *command;* the resulting output then replaces the input lines.

($) =

Prints the line number of the addressed line. The current line is unchanged.

(. , .) > *count flags*
(. , .) < *count flags*

Perform intelligent shifting on the specified lines; < shifts left and > shift right. The quantity of shift is determined by the *shiftwidth* option and the repetition of the specification character. Only white space (blanks and tabs) is shifted; no non-white characters are discarded in a left-shift. The current line becomes the last line which changed due to the shifting.

^D

An end-of-file from a terminal input scrolls through the file. The *scroll* option specifies the size of the scroll, normally a half screen of text.

(.+1 , .+1)
(.+1 , .+1) |

An address alone causes the addressed lines to be printed. A blank line prints the next line in the file.

* Forms 'z=' and 'z↑' also exist; 'z=' places the current line in the center, surrounds it with lines of '−' characters and leaves the current line at this line. The form 'z↑' prints the window before 'z−' would. The characters '+', '↑' and '−' may be repeated for cumulative effect. On some v2 editors, no *type* may be given.

(. , .) & *options count flags*

Repeats the previous *substitute* command.

(. , .) ~ *options count flags*

Replaces the previous regular expression with the previous replacement pattern from a substitution.

8. Regular expressions and substitute replacement patterns

8.1. Regular expressions

A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. *Ex* remembers two previous regular expressions: the previous regular expression used in a *substitute* command and the previous regular expression used elsewhere (referred to as the previous *scanning* regular expression.) The previous regular expression can always be referred to by a null *re*, e.g. '//' or '??'.

8.2. Magic and nomagic

The regular expressions allowed by *ex* are constructed in one of two ways depending on the setting of the *magic* option. The *ex* and *vi* default setting of *magic* gives quick access to a powerful set of regular expression metacharacters. The disadvantage of *magic* is that the user must remember that these metacharacters are *magic* and precede them with the character '\' to use them as "ordinary" characters. With *nomagic,* the default for *edit,* regular expressions are much simpler, there being only two metacharacters. The power of the other metacharacters is still available by preceding the (now) ordinary character with a '\'. Note that '\' is thus always a metacharacter.

The remainder of the discussion of regular expressions assumes that that the setting of this option is *magic.*†

8.3. Basic regular expression summary

The following basic constructs are used to construct *magic* mode regular expressions.

char	An ordinary character matches itself. The characters '↑' at the beginning of a line, '$' at the end of line, '*' as any character other than the first, '.', '\', '[', and '~' are not ordinary characters and must be escaped (preceded) by '\' to be treated as such.
↑	At the beginning of a pattern forces the match to succeed only at the beginning of a line.
$	At the end of a regular expression forces the match to succeed only at the end of the line.
.	Matches any single character except the new-line character.
\<	Forces the match to occur only at the beginning of a "variable" or "word"; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.
\>	Similar to '\<', but matching the end of a "variable" or "word", i.e. either the end of the line or before character which is neither a letter, nor a digit, nor the underline character.

† To discern what is true with *nomagic* it suffices to remember that the only special characters in this case will be '↑' at the beginning of a regular expression, '$' at the end of a regular expression, and '\'. With *nomagic* the characters '~' and '&' also lose their special meanings related to the replacement pattern of a substitute.

[string] Matches any (single) character in the class defined by *string*. Most characters in *string* define themselves. A pair of characters separated by '−' in *string* defines the set of characters collating between the specified lower and upper bounds, thus '[a−z]' as a regular expression matches any (single) lower-case letter. If the first character of *string* is an '↑' then the construct matches those characters which it otherwise would not; thus '[↑a−z]' matches anything but a lower-case letter (and of course a newline). To place any of the characters '↑', '[', or '−' in *string* you must escape them with a preceding '\'.

8.4. Combining regular expression primitives

The concatenation of two regular expressions matches the leftmost and then longest string which can be divided with the first piece matching the first regular expression and the second piece matching the second. Any of the (single character matching) regular expressions mentioned above may be followed by the character '*' to form a regular expression which matches any number of adjacent occurrences (including 0) of characters matched by the regular expression it follows.

The character '~' may be used in a regular expression, and matches the text which defined the replacement part of the last *substitute* command. A regular expression may be enclosed between the sequences '\(' and '\)' with side effects in the *substitute* replacement patterns.

8.5. Substitute replacement patterns

The basic metacharacters for the replacement pattern are '&' and '~'; these are given as '\&' and '\~' when *nomagic* is set. Each instance of '&' is replaced by the characters which the regular expression matched. The metacharacter '~' stands, in the replacement pattern, for the defining text of the previous replacement pattern.

Other metasequences possible in the replacement pattern are always introduced by the escaping character '\'. The sequence '*n*' is replaced by the text matched by the *n*-th regular subexpression enclosed between '\(' and '\)'.† The sequences '\u' and '\l' cause the immediately following character in the replacement to be converted to upper- or lower-case respectively if this character is a letter. The sequences '\U' and '\L' turn such conversion on, either until '\E' or '\e' is encountered, or until the end of the replacement pattern.

9. Option descriptions

autoindent, ai default: noai

Can be used to ease the preparation of structured program text. At the beginning of each *append*, *change* or *insert* command or when a new line is *opened* or created by an *append*, *change*, *insert*, or *substitute* operation within *open* or *visual* mode, *ex* looks at the line being appended after, the first line changed or the line inserted before and calculates the amount of white space at the start of the line. It then aligns the cursor at the level of indentation so determined.

If the user then types lines of text in, they will continue to be justified at the displayed indenting level. If more white space is typed at the beginning of a line, the following line will start aligned with the first non-white character of the previous line. To back the cursor up to the preceding tab stop one can hit **^D**. The tab stops going backwards are defined at multiples of the *shiftwidth* option. You *cannot* backspace over the indent, except by sending an end-of-file with a **^D**.

† When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of '\(' starting from the left.

Specially processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the *autoindent* is discarded.) Also specially processed in this mode are lines beginning with an '↑' and immediately followed by a **^D**. This causes the input to be repositioned at the beginning of the line, but retaining the previous indent for the next line. Similarly, a '0' followed by a **^D** repositions at the beginning but without retaining the previous indent.

Autoindent doesn't happen in *global* commands or when the input is not a terminal.

autoprint, ap default: ap

Causes the current line to be printed after each *delete*, *copy*, *join*, *move*, *substitute*, *t*, *undo* or shift command. This has the same effect as supplying a trailing 'p' to each such command. *Autoprint* is suppressed in globals, and only applies to the last of many commands on a line.

autowrite, aw default: noaw

Causes the contents of the buffer to be written to the current file if you have modified it and give a *next, rewind, stop, tag,* or *!* command, or a **^↑** (switch files) or **^]** (tag goto) command in *visual.* Note, that the *edit* and *ex* commands do **not** autowrite. In each case, there is an equivalent way of switching when autowrite is set to avoid the *autowrite* (*edit* for *next*, *rewind!* for .I rewind , *stop!* for *stop*, *tag!* for *tag*, *shell* for *!*, and **:e #** and a **:ta!** command from within *visual).*

beautify, bf default: nobeautify

Causes all control characters except tab, newline and form-feed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. *Beautify* does not apply to command input.

directory, dir default: dir=/tmp

Specifies the directory in which *ex* places its buffer file. If this directory in not writable, then the editor will exit abruptly when it fails to be able to create its buffer there.

edcompatible default: noedcompatible

Causes the presence of absence of **g** and **c** suffixes on substitute commands to be remembered, and to be toggled by repeating the suffices. The suffix **r** makes the substitution be as in the **~** command, instead of like *&*. ‡‡

errorbells, eb default: noeb

Error messages are preceded by a bell.* If possible the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

hardtabs, ht default: ht=8

Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).

ignorecase, ic default: noic

‡‡ Version 3 only.

* Bell ringing in *open* and *visual* on errors is not suppressed by setting *noeb.*

All upper case characters in the text are mapped to lower case in regular expression matching. In addition, all upper case characters in regular expressions are mapped to lower case except in character class specifications.

lisp default: nolisp

Autoindent indents appropriately for *lisp* code, and the **( ) { } [[** and **]]** commands in *open* and *visual* are modified to have meaning for *lisp.*

list default: nolist

All printed lines will be displayed (more) unambiguously, showing tabs and end-of-lines as in the *list* command.

magic default: magic for *ex* and *vi*†

If *nomagic* is set, the number of regular expression metacharacters is greatly reduced, with only '↑' and '$' having special effects. In addition the metacharacters '~' and '&' of the replacement pattern are treated as normal characters. All the normal metacharacters may be made *magic* when *nomagic* is set by preceding them with a '\'.

mesg default: mesg

Causes write permission to be turned off to the terminal while you are in visual mode, if *nomesg* is set. ‡‡

number, nu default: nonumber

Causes all output lines to be printed with their line numbers. In addition each input line will be prompted for by supplying the line number it will have.

open default: open

If *noopen*, the commands *open* and *visual* are not permitted. This is set for *edit* to prevent confusion resulting from accidental entry to open or visual mode.

optimize, opt default: optimize

Throughput of text is expedited by setting the terminal to not do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.

paragraphs, para default: para=IPLPPPQPP LIbp

Specifies the paragraphs for the **{** and **}** operations in *open* and *visual.* The pairs of characters in the option's value are the names of the macros which start paragraphs.

prompt default: prompt

Command mode input is prompted for with a ':'.

redraw default: noredraw

The editor simulates (using great amounts of output), an intelligent terminal on a dumb terminal (e.g. during insertions in *visual* the characters to the right of the cursor position are refreshed as each input character is typed.) Useful only at very high speed.

---

† *Nomagic* for *edit.*
‡‡ Version 3 only.

remap default: remap

If on, macros are repeatedly tried until they are unchanged. ‡‡ For example, if **o** is mapped to **O**, and **O** is mapped to **I**, then if *remap* is set, **o** will map to **I**, but if *noremap* is set, it will map to **O**.

report default: report=5†

Specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines will provide feedback as to the scope of its changes. For commands such as *global*, *open*, *undo*, and *visual* which have potentially more far reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus notification is suppressed during a *global* command on the individual commands performed.

scroll default: scroll=½ window

Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command mode *z* command (double the value of *scroll*).

sections default: sections=SHNHH HU

Specifies the section macros for the **[[** and **]]** operations in *open* and *visual*. The pairs of characters in the options's value are the names of the macros which start paragraphs.

shell, sh default: sh=/bin/sh

Gives the path name of the shell forked for the shell escape command '!', and by the *shell* command. The default is taken from SHELL in the environment, if present.

shiftwidth, sw default: sw=8

Gives the width a software tab stop, used in reverse tabbing with **^D** when using *autoindent* to append text, and by the shift commands.

showmatch, sm default: nosm

In *open* and *visual* mode, when a **)** or **}** is typed, move the cursor to the matching **(** or **{** for one second if this matching character is on the screen. Extremely useful with *lisp*.

slowopen, slow terminal dependent

Affects the display algorithm used in *visual* mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent. See *An Introduction to Display Editing with Vi* for more details.

tabstop, ts default: ts=8

The editor expands tabs in the input file to be on *tabstop* boundaries for the purposes of display.

taglength, tl default: tl=0

Tags are not significant beyond this many characters. A value of zero (the default) means that all characters are significant.

---

‡‡ Version 3 only.

† 2 for *edit*.

tags default: tags=tags /usr/lib/tags

A path of files to be used as tag files for the *tag* command. ‡‡ A requested tag is searched for in the specified files, sequentially. By default (even in version 2) files called **tags** are searched for in the current directory and in /usr/lib (a master file for the entire system.)

term from environment TERM

The terminal type of the output device.

terse default: noterse

Shorter error diagnostics are produced for the experienced user.

warn default: warn

Warn if there has been '[No write since last change]' before a '!' command escape.

window default: window=speed dependent

The number of lines in a text window in the *visual* command. The default is 8 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.

w300, w1200, w9600

These are not true options but set **window** only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule.

wrapscan, ws default: ws

Searches using the regular expressions in addressing will wrap around past the end of the file.

wrapmargin, wm default: wm=0

Defines a margin for automatic wrapover of text during input in *open* and *visual* modes. See *An Introduction to Text Editing with Vi* for details.

writeany, wa default: nowa

Inhibit the checks normally made before *write* commands, allowing a write to any file which the system protection mechanism will allow.

10. Limitations

Editor limits that the user is likely to encounter are as follows: 1024 characters per line, 256 characters per global command list, 128 characters per file name, 128 characters in the previous inserted and deleted text in *open* or *visual,* 100 characters in a shell escape command, 63 characters in a string valued option, and 30 characters in a tag name, and a limit of 250000 lines in the file is silently enforced.

The *visual* implementation limits the number of macros defined with map to 32, and the total number of characters in macros to be less than 512.

Acknowledgments. Chuck Haley contributed greatly to the early development of *ex.* Bruce Englar encouraged the redesign which led to *ex* version 1. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and Unix systems.

‡‡ Version 3 only.

Ex changes — Version 3.1 to 3.5

This update describes the new features and changes which have been made in converting from version 3.1 to 3.5 of *ex.* Each change is marked with the first version where it appeared.

Update to Ex Reference Manual

Command line options

3.4 A new command called *view* has been created. *View* is just like *vi* but it sets *readonly.*

3.4 The encryption code from the v7 editor is now part of *ex.* You can invoke *ex* with the −x option and it will ask for a key, as *ed.* The *ed x* command (to enter encryption mode from within the editor) is not available. This feature may not be available in all instances of *ex* due to memory limitations.

Commands

3.4 Provisions to handle the new process stopping features of the Berkeley TTY driver have been added. A new command, *stop,* takes you out of the editor cleanly and efficiently, returning you to the shell. Resuming the editor puts you back in command or visual mode, as appropriate. If *autowrite* is set and there are outstanding changes, a write is done first unless you say "stop!".

3.4 A

```
:vi <file>
```

command from visual mode is now treated the same as a

```
:edit <file>     or     :ex <file>
```

command. The meaning of the *vi* command from *ex* command mode is not affected.

3.3 A new command mode command *xit* (abbreviated *x*) has been added. This is the same as *wq* but will not bother to write if there have been no changes to the file.

Options

3.4 A read only mode now lets you guarantee you won't clobber your file by accident. You can set the on/off option *readonly* (*ro*), and writes will fail unless you use an ! after the write. Commands such as *x, ZZ,* the autowrite option, and in general anything that writes is affected. This option is turned on if you invoke ex with the −R flag.

3.4 The *wrapmargin* option is now usable. The way it works has been completely revamped. Now if you go past the margin (even in the middle of a word) the entire word is erased and rewritten on the next line. This changes the semantics of the number given to wrapmargin. 0 still means off. Any other number is still a distance from the right edge of the screen, but this location is now the right edge of the area where wraps can take place, instead of the left edge. Wrapmargin now behaves much like fill/nojustify mode in *nroff.*

3.3 The options *w300, w1200,* and *w9600* can be set. They are synonyms for *window,* but only apply at 300, 1200, or 9600 baud, respectively. Thus you can specify you want a 12 line window at 300 baud and a 23 line window at 1200 baud in your EXINIT with

```
:set w300=12 w1200=23
```

3.3 The new option *timeout* (default on) causes macros to time out after one second. Turn it off and they will wait forever. This is useful if you want multi character macros, but if your terminal sends escape sequences for arrow keys, it will be necessary to hit escape twice to get a beep.

3.3 The new option *remap* (default on) causes the editor to attempt to map the result of a macro mapping again until the mapping fails. This makes it possible, say, to map q to # and #1 to something else and get q1 mapped to something else. Turning it off makes it possible to map ^L to l and map ^R to ^L without having ^R map to l.

3.3 The new (string) valued option *tags* allows you to specify a list of tag files, similar to the "path" variable of csh. The files are separated by spaces (which are entered preceded by a backslash) and are searched left to right. The default value is "tags /usr/lib/tags", which has the same effect as before. It is recommended that "tags" always be the first entry. On Ernie CoVax, /usr/lib/tags contains entries for the system defined library procedures from section 3 of the manual.

Environment enquiries

3.4 The editor now adopts the convention that a null string in the environment is the same as not being set. This applies to TERM, TERMCAP, and EXINIT.

Vi Tutorial Update

Deleted features

3.3 The "q" command from visual no longer works at all. You must use "Q" to get to ex command mode. The "q" command was deleted because of user complaints about hitting it by accident too often.

3.5 The provisions for changing the window size with a numeric prefix argument to certain visual commands have been deleted. The correct way to change the window size is to use the z command, for example z5<cr> to change the window to 5 lines.

3.3 The option "mapinput" is dead. It has been replaced by a much more powerful mechanism: ":map!".

Change in default option settings

3.3 The default window sizes have been changed. At 300 baud the window is now 8 lines (it was 1/2 the screen size). At 1200 baud the window is now 16 lines (it was 2/3 the screen size, which was usually also 16 for a typical 24 line CRT). At 9600 baud the window is still the full screen size. Any baud rate less than 1200 behaves like 300, any over 1200 like 9600. This change makes *vi* more usable on a large screen at slow speeds.

Vi commands

3.3 The command "ZZ" from vi is the same as ":x<cr>". This is the recommended way to leave the editor. Z must be typed twice to avoid hitting it accidently.

3.4 The command ^Z is the same as ":stop<cr>". Note that if you have an arrow key that sends ^Z the stop function will take priority over the arrow function. If you have your "susp" character set to something besides ^Z, that key will be honored as well.

3.3 It is now possible from visual to string several search expressions together separated by semicolons the same as command mode. For example, you can say

```
/foo/;/bar
```

from visual and it will move to the first "bar" after the next "foo". This also works within one line.

3.3 ^R is now the same as ^L on terminals where the right arrow key sends ^L (This includes the Televideo 912/920 and the ADM 31 terminals.)

3.4 The visual page motion commands ^F and ^B now treat any preceding counts as number of pages to move, instead of changes to the window size. That is, 2^F moves forward 2 pages.

Macros

3.3 The ‘‘mapinput’’ mechanism of version 3.1 has been replaced by a more powerful mechanism. An ‘‘!’’ can follow the word ‘‘**map**’’ in the *map* command. **Map!**’ed macros only apply during input mode, while **map**’ed macros only apply during command mode. Using ‘‘**map**’’ or ‘‘**map!**’’ by itself produces a listing of macros in the corresponding mode.

3.4 A word abbreviation mode is now available. You can define abbreviations with the *abbreviate* command

```
:abbr foo find outer otter
```

which maps ‘‘foo’’ to ‘‘find outer otter’’. Abbreviations can be turned off with the *unabbreviate* command. The syntax of these commands is identical to the *map* and *unmap* commands, except that the ! forms do not exist. Abbreviations are considered when in visual input mode only, and only affect whole words typed in, using the conservative definition. (Thus ‘‘foobar’’ will not be mapped as it would using ‘‘map!’’) Abbreviate and unabbreviate can be abbreviated to ‘‘ab’’ and ‘‘una’’, respectively.

SED — A Non-interactive Text Editor

Lee E. McMahon

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Sed is a non-interactive context editor that runs on the UNIX† operating system. *Sed* is designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

This memorandum constitutes a manual for users of *sed.*

August 15, 1978

†UNIX is a Trademark of Bell Laboratories.

SED — A Non-interactive Text Editor

Lee E. McMahon

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

Sed is a non-interactive context editor designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode;
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

Since only a few lines of the input reside in core at one time, and no temporary files are used, the effective size of file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to *sed* as a command file. For complex edits, this saves considerable typing, and its attendant errors. *Sed* running from a command file is much more efficient than any interactive editor known to the author, even if that editor can be driven by a pre-written script.

The principal loss of functions compared to an interactive editor are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

Sed is a lineal descendant of the UNIX editor, *ed*. Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed;* even confirmed users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without reading Sections 2 and 3 of this document. The most striking family resemblance between the two editors is in the class of patterns ('regular expressions') they recognize; the code for matching patterns is copied almost verbatim from the code for *ed*, and the description of regular expressions in Section 2 is copied almost verbatim from the UNIX Programmer's Manual[1]. (Both code and description were written by Dennis M. Ritchie.)

1. Overall Operation

Sed by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line; see Section 1.1 below.

The general format of an editing command is:

```
[address1,address2][function][arguments]
```

One or both addresses may be omitted; the format of addresses is given in Section 2. Any number of blanks or tabs may separate the addresses from the function. The function must be present; the available commands are discussed in Section 3. The arguments may be required or optional, according to which function is given; again, they are discussed in Section 3 under each individual function.

Tab characters and spaces at the beginning of lines are ignored.

1.1. Command-line Flags

Three flags are recognized on the command line:

-n: tells *sed* not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions (see Section 3.3);

-e: tells *sed* to take the next argument as an editing command;

-f: tells *sed* to take the next argument as a file name; the file should contain editing commands, one to a line.

1.2. Order of Application of Editing Commands

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands, *t* and *b* (see Section 3). Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

1.3. Pattern-space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the *N* command (Section 3.6.).

1.4. Examples

Examples are scattered throughout the text. Except where otherwise noted, the examples all assume the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

(In no case is the output of the *sed* commands to be considered an improvement on Coleridge.)

Example:

The command

```
2q
```

will quit after copying the first two lines of the input. The output will be:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

2. ADDRESSES: Selecting lines for editing

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands with curly braces ('{ }')(Sec. 3.6.).

2.1. Line-number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches (selects) the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character $ matches the last line of the last input file.

2.2. Context Addresses

A context address is a pattern ('regular expression') enclosed in slashes ('/'). The regular expressions recognized by *sed* are constructed as follows:

1) An ordinary character (not one of those discussed below) is a regular expression, and matches that character.
2) A circumflex '^' at the beginning of a regular expression matches the null character at the beginning of a line.
3) A dollar-sign '$' at the end of a regular expression matches the null character at the end of a line.
4) The characters '\n' match an imbedded newline character, but not the newline at the end of the pattern space.
5) A period '.' matches any character except the terminal newline of the pattern space.
6) A regular expression followed by an asterisk '*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.
7) A string of characters in square brackets '[ ]' matches any character in the string, and no others. If, however, the first character of the string is circumflex '^', the regular expression matches any character *except* the characters in the string and the terminal newline of the pattern space.
8) A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
9) A regular expression between the sequences '\(' and '\)' is identical in effect to the unadorned regular expression, but has side-effects which are described under the *s* command below and specification 10) immediately below.
10) The expression '\d' means the same string of characters matched by an expression enclosed in '\(' and '\)' earlier in the same pattern. Here *d* is a single digit; the string specified is that *beginning with the d*th occurrence of '\(' counting from the left. For example, the expression '^\(.*\)\1' matches a line beginning with two repeated occurrences of the same string.
11) The null regular expression standing alone (e.g., '//') is equivalent to the last regular expression compiled.

To use one of the special characters (^ $. * [ ] \ /) as a literal (to match an occurrence of itself in the input), precede the special character by a backslash '\'.

For a context address to 'match' the input requires that the whole pattern within the address match some portion of the pattern space.

2.3. Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines which match that address.

If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and including) the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address, and the process is repeated.

Two addresses are separated by a comma.

Examples:

/an/	matches lines 1, 3, 4 in our sample text
/an.*an/	matches line 1
/^an/	matches no lines
/./	matches all lines
/\./	matches line 5
/r*an/	matches lines 1,3, 4 (number = zero!)
/\(an\).*\1/	matches line 1

3. FUNCTIONS

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is given enclosed in parentheses, then the single character function name, possible arguments enclosed in angles (< >), an expanded English translation of the single-character name, and finally a description of what each function does. The angles around the arguments are *not* part of the argument, and should not be typed in actual editing commands.

3.1. Whole-line Oriented Functions

(2)d -- delete lines

The *d* function deletes from the file (does not write to the output) all those lines matched by its address(es).

It also has the side effect that no further commands are attempted on the corpse of a deleted line; as soon as the *d* function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

(2)n -- next line

The *n* function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the *n* command.

(1)a\
<text> -- append lines

The *a* function causes the argument <text> to be written to the output after the line matched by its address. The *a* command is inherently multi-line; *a* must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character ('\') immediately preceding the newline. The <text> argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash).

Once an *a* function is successfully executed, <text> will be written to the output regardless of what later commands do to the line which triggered it. The triggering line may be deleted entirely; <text> will still be written to the output.

The <text> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

(1)i\
<text> -- insert lines

The *i* function behaves identically to the *a* function, except that <text> is written to the output *before* the matched line. All other comments about the *a* function apply to the *i* function as well.

(2)c\
<text> -- change lines

The *c* function deletes the lines selected by its address(es), and replaces them with the lines in <text>. Like *a* and *i*, *c* must be followed by a newline hidden by a backslash; and interior new lines in <text> must be hidden by backslashes.

The *c* command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of <text> is written to the output, *not* one copy per line deleted. As with *a* and *i*, <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a *c* function, no further commands are attempted on the corpse.

If text is appended after a line by *a* or *r* functions, and the line is subsequently changed, the text inserted by the *c* function will be placed *before* the text of the *a* or *r* functions. (The *r* function is described in Section 3.4.)

Note: Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in *sed* commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

Example:

The list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n        n
i\       c\
XXXX     XXXX
d
```

3.2. Substitute Function

One very important function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> -- substitute

The *s* function replaces *part* of a line (selected by <pattern>) with <replacement>. It can best be read:

Substitute for <pattern>, <replacement>

The <pattern> argument contains a pattern, exactly like the patterns in addresses (see 2.2 above). The only difference between <pattern> and a context address is that the context address must be delimited by slash ('/') characters; <pattern> may be delimited by any character other than space or newline.

By default, only the first string matched by <pattern> is replaced, but see the *g* flag below.

The <replacement> argument begins immediately after the second delimiting character of <pattern>, and must be followed immediately by another instance of the delimiting character. (Thus there are exactly *three* instances of the delimiting character.)

The <replacement> is not a pattern, and the characters which are special in patterns do not have special meaning in <replacement>. Instead, other characters are special:

&   is replaced by the string matched by <pattern>

\d (where *d* is a single digit) is replaced by the *d*th substring matched by parts of <pattern> enclosed in '\\(' and '\\)'. If nested substrings occur in <pattern>, the *d*th is determined by counting opening delimiters ('\\(').

As in patterns, special characters may be made literal by preceding them with backslash ('\\').

The <flags> argument may contain the following flags:

g -- substitute <replacement> for all (non-overlapping) instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters; characters put into the line from <replacement> are not rescanned.

p -- print the line if a successful replacement was done. The *p* flag causes the line to be written to the output if and only if a substitution was actually made by the *s* function. Notice that if several *s* functions, each followed by a *p* flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

w <filename> -- write the line to a file if a successful replacement was done. The *w* flag causes lines which are actually substituted by the *s* function to be written to a file named by <filename>. If <filename> exists before *sed* is run, it is overwritten; if not, it is created.

A single space must separate *w* and <filename>.

The possibilities of multiple, somewhat different copies of one input line being written are the same as for *p*.

A maximum of 10 different file names may be mentioned after *w* flags and *w* functions (see below), combined.

Examples:

The following command, applied to our standard input,

```
s/to/by/w changes
```

produces, on the standard output:

```
In Xanadu did Kubhla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and, on the file 'changes':

```
Through caverns measureless by man
Down by a sunless sea.
```

If the nocopy option is in effect, the command:

```
s/[.,;?:]/*P&*/gp
```

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

Finally, to illustrate the effect of the *g* flag, the command:

```
/X/s/an/AN/p
```

produces (assuming nocopy mode):

```
In XANadu did Kubhla Khan
```

and the command:

```
/X/s/an/AN/gp
```

produces:

```
In XANadu did Kubhla KhAN
```

3.3. Input-output Functions

(2)p -- print

The print function writes the addressed lines to the standard output file. They are written at the time the *p* function is encountered, regardless of what succeeding editing commands may do to the lines.

(2)w <filename> -- write on <filename>

The write function writes the addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them.

Exactly one space must separate the *w* and <filename>.

A maximum of ten different files may be mentioned in write functions and *w* flags after *s* functions, combined.

(1)r <filename> -- read the contents of a file

The read function reads the contents of <filename>, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If *r* and *a* functions are executed on the same line, the text from the *a* functions and the *r* functions is written to the output in the order that the functions are executed.

Exactly one space must separate the *r* and <filename>. If a file mentioned by a *r* function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

NOTE: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in *w* functions or flags; that number is reduced by one if any *r* functions are present. (Only one read file is open at one time.)

Examples

Assume that the file 'note1' has the following contents:

```
                Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson
                and most eminent successor of Genghiz (Chingiz) Khan, and founder of the
                Mongol dynasty in China.
```

Then the following command:

```
        /Kubla/r note1
```

produces:

```
        In Xanadu did Kubla Khan
                Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson
                and most eminent successor of Genghiz (Chingiz) Khan, and founder of the
                Mongol dynasty in China.
        A stately pleasure dome decree:
        Where Alph, the sacred river, ran
        Through caverns measureless to man
        Down to a sunless sea.
```

3.4. Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing imbedded newlines; they are intended principally to provide pattern matches across lines in the input.

(2)N -- Next line

The next input line is appended to the current line in the pattern space; the two input lines are separated by an imbedded newline. Pattern matches may extend across the imbedded newline(s).

(2)D -- Delete first part of the pattern space

Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2)P -- Print first part of the pattern space

Print up to and including the first newline in the pattern space.

The *P* and *D* functions are equivalent to their lower-case counterparts if there are no imbedded newlines in the pattern space.

3.5. Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)h -- hold pattern space

> The *h* functions copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

(2)H -- Hold pattern space

> The *H* function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

(2)g -- get contents of hold area

> The *g* function copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).

(2)G -- Get contents of hold area

> The *G* function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

(2)x -- exchange

> The exchange command interchanges the contents of the pattern space and the hold area.

Example

The commands

```
1h
1s/ did.*//
1x
G
s/\n/ :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan :In Xanadu
A stately pleasure dome decree: :In Xanadu
Where Alph, the sacred river, ran :In Xanadu
Through caverns measureless to man :In Xanadu
Down to a sunless sea. :In Xanadu
```

3.6. Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

(2)! -- Don't

> The *Don't* command causes the next command (written on the same line), to be applied to all and only those input lines *not* selected by the adress part.

(2){ -- Grouping

> The grouping command '{' causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the '{' or on the next line.

The group of commands is terminated by a matching '}' standing on a line by itself.

Groups can be nested.

(0):<label> -- place a label

The label function marks a place in the list of editing commands which may be referred to by *b* and *t* functions. The <label> may be any sequence of eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2)b<label> -- branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A *b* function with no <label> is taken to be a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line.

(2)t<label> -- test substitutions

The *t* function tests whether *any* successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

1) reading a new input line, or
2) executing a *t* function.

3.7. Miscellaneous Functions

(1)= -- equals

The = function writes to the standard output the line number of the line matched by its address.

(1)q -- quit

The *q* function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

Reference

[1] Ken Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual.* Bell Laboratories, 1978.

Awk — A Pattern Scanning and Processing Language
(Second Edition)

Alfred V. Aho

Brian W. Kernighan

Peter J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Awk is a programming language whose basic operation is to search a set of files for patterns, and to perform specified actions upon lines or fields of lines which contain instances of those patterns. *Awk* makes certain data selection and transformation operations easy to express; for example, the *awk* program

```
length > 72
```

prints all input lines whose length exceeds 72 characters; the program

```
NF % 2 == 0
```

prints all lines with an even number of fields; and the program

```
{ $1 = log($1); print }
```

replaces the first field of each line by its logarithm.

Awk patterns may include arbitrary boolean combinations of regular expressions and of relational operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns, as well as arithmetic and string expressions and assignments, **if-else**, **while**, **for** statements, and multiple output streams.

This report contains a user's guide, a discussion of the design and implementation of *awk*, and some timing statistics.

September 1, 1978

Awk — A Pattern Scanning and Proce sing Language (Second Edition)

Alfred V. Aho

Brian W. Kernighan

Peter J. Weinberger

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

The basic operation of *awk* is to scan a set of input lines in order, searching for lines which match any of a set of patterns which the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern.

Readers familiar with the UNIX† program *grep*[1] will recognize the approach, although in *awk* the patterns may be more general than in *grep*, and the actions allowed are more involved than merely printing the matching line. For example, the *awk* program

```
{print $3, $2}
```

prints the third and second columns of a table in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev   { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

1.1. Usage

The command

```
awk  program  [files]
```

executes the *awk* commands in the string program on the set of named files, or on the standard input if there are no files. The statements can also be placed in a file pfile, and executed by the command

```
awk  -f pfile  [files]
```

1.2. Program Structure

An *awk* program is a sequence of statements of the form:

```
pattern     { action }
pattern     { action }
...
```

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

1.3. Records and Fields

Awk input is divided into "records" terminated by a record separator. The default record separator is a newline, so by default *awk* processes its input a line at a time. The number of the current record is available in a variable named NR.

Each input record is considered to be divided into "fields." Fields are normally separated by white space — blanks or tabs — but the input field separator may be changed, as described below. Fields are referred to as $1, $2, and so forth, where $1 is the first field, and $0 is the whole input record itself. Fields may

†UNIX is a Trademark of Bell Laboratories.

be assigned to. The number of fields in the current record is available in a variable named NF.

The variables FS and RS refer to the input field and record separators; they may be changed at any time to any single character. The optional command-line argument −F*c* may also be used to set FS to the character *c*.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable FILENAME contains the name of the current input file.

1.4. Printing

An action may have no pattern, in which case the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the *awk* command print. The *awk* program

```
{ print }
```

prints each record, thus copying the input to the output intact. More useful is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

runs the first and second fields together.

The predefined variables NF and NR can be used; for example

```
{ print NR, NF, $0 }
```

prints each record preceded by the record number and the number of fields.

Output may be diverted to multiple files; the program

```
{ print $1 >"foo1"; print $2 >"foo2" }
```

writes the first field, $1, on the file foo1, and the second field on file foo2. The >> notation can also be used:

```
print $1 >>"foo"
```

appends the output to the file foo. (In each case, the output files are created if necessary.) The file name can be a variable or a field as well as a constant; for example,

```
print $1 >$2
```

uses the contents of field 2 as a file name.

Naturally there is a limit on the number of output files; currently it is 10.

Similarly, output can be piped into another process (on UNIX only); for instance,

```
print | "mail bwk"
```

mails the output to bwk.

The variables OFS and ORS may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the print statement.

Awk also provides the printf statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in format and prints them. For example,

```
printf "%8.2f  %10ld\n", $1, $2
```

prints $1 as a floating point number 8 digits wide, with two after the decimal point, and $2 as a 10-digit long decimal number, followed by a newline. No output separators are produced automatically; you must add them yourself, as in this example. The version of printf is identical to that used with C.[2]

2. Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

2.1. BEGIN and END

The special pattern BEGIN matches the beginning of the input, before the first record is read. The pattern END matches the end of the input, after the last record has been processed. BEGIN and END thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

```
BEGIN    { FS = ":" }
... rest of program ...
```

Or the input lines may be counted by

```
END  { print NR }
```

If BEGIN is present, it must be the first pattern; END must be the last if used.

2.2. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete *awk* program which will print all lines which contain any occurrence of the name "smith". If a line contains "smith" as part of a larger word, it will also be printed, as in

```
blacksmithing
```

Awk regular expressions include the regular expression forms found in the UNIX text editor *ed*[1] and *grep* (without back-referencing). In addition, *awk* allows parentheses for grouping, | for alternatives, + for "one or more", and ? for "zero or one", all as in *lex*. Character classes may be abbreviated: [a−zA−Z0−9] is the set of all letters and digits. As an example, the *awk* program

```
/[Aa]ho|[Ww]einberger|[Kk]ernighan/
```

will print all lines which contain any of the names "Aho," "Weinberger" or "Kernighan," whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in *ed* and *sed*. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn of the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

```
/\/.*\//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) with the operators ~ and !~. The program

```
$1 ~ /[jJ]ohn/
```

prints all lines where the first field matches "john" or "John." Notice that this will also match "Johnson", "St. Johnsbury", and so on. To restrict it to exactly [jJ]ohn, use

```
$1 ~ /^[jJ]ohn$/
```

The caret ^ refers to the beginning of a line or field; the dollar sign $ refers to the end.

2.3. Relational Expressions

An *awk* pattern can be a relational expression involving the usual relational operators <, <=, ==, !=, >=, and >. An example is

```
$2 > $1 + 100
```

which selects lines where the second field is at least 100 greater than the first field. Similarly,

```
NF % 2 == 0
```

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
$1 >= "s"
```

selects lines that begin with an **s**, **t**, **u**, etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

will perform a string comparison.

2.4. Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example,

```
$1 >= "s" && $1 < "t" && $1 != "smith"
```

selects lines where the first field begins with "s", but is not "smith". && and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

2.5. Pattern Ranges

The "pattern" that selects an action may also consist of two patterns separated by a comma, as in

```
pat1, pat2     { ... }
```

In this case, the action is performed for each line between an occurrence of **pat1** and the next occurrence of **pat2** (inclusive). For example,

```
/start/, /stop/
```

prints all lines between **start** and **stop**, while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

3. Actions

An *awk* action is a sequence of action statements terminated by newlines or semicolors. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

3.1. Built-in Functions

Awk provides a "length" function to compute the length of a string of characters. This program prints each record, preceded by its length:

```
{print length, $0}
```

length by itself is a "pseudo-variable" which yields the length of the current record; length(argument) is a function which yields the length of its argument, as in the equivalent

```
{print length($0), $0}
```

The argument may be any expression.

Awk also provides the arithmetic functions sqrt, log, exp, and int, for square root, base *e* logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function substr(s, m, n) produces the substring of s that begins at position m (origin 1) and is at most n characters long. If n is omitted, the substring goes to the end of s. The function index(s1, s2) returns the position where the string s2 occurs in s1, or zero if it does not.

The function sprintf(f, e1, e2, ...) produces the value of the expressions e1, e2, etc., in the printf format specified by f. Thus, for example,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets x to the string produced by formatting the values of $1 and $2.

3.2. Variables, Expressions, and Assignments

Awk variables take on numeric (floating point) or string values according to context. For example, in

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns 7 to x. Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most BEGIN sections. For example, the sums of the first two fields can be computed by

```
    { s1 += $1; s2 += $2 }
END { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, −, *, /, and % (mod). The C increment ++ and decrement −− operators are also available, and so are the assignment operators +=, −=, *=, /=, and %=. These operators may all be used in expressions.

3.3. Field Variables

Fields in *awk* share essentially all of the properties of variables — they may be used in arithmetic or string operations, and may be assigned to. Thus one can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
      $3 = "too big"
  print
}
```

which replaces the third field by "too big" when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

```
n = split(s, array, sep)
```

splits the the string s into array[1], ..., array[n]. The number of elements found is returned. If the sep argument is provided, it is used as the field separator; otherwise FS is used as the separator.

3.4. String Concatenation

Strings may be concatenated. For example

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a print statement,

```
print $1 " is " $2
```

prints the two fields separated by " is ". Variables and numeric expressions may also appear in concatenations.

3.5. Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have *any* non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the NR-th element of the array x. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the *awk* program

```
    { x[NR] = $0 }
END { ... program ... }
```

The first action merely records each input line in the array x.

Array elements may be named by non-numeric values, which gives *awk* a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like apple, orange, etc. Then the program

```
/apple/   { x["apple"]++ }
/orange/  { x["orange"]++ }
END       { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

3.6. Flow-of-Control Statements

Awk provides the basic flow-of-control statements if-else, while, for, and statement grouping with braces, as in C. We showed the if statement in section 3.3 without describing it. The condition in parentheses is evaluated; if it is true, the statement following the if is done. The else part is optional.

The while statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
     print $i
     ++i
}
```

The for statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
     print $i
```

does the same job as the while statement above.

There is an alternate form of the for statement which is suited for accessing the elements of an associative array:

```
for (i in array)
     statement
```

does *statement* with i set in turn to each element of array. The elements are accessed in an apparently random order. Chaos will ensue if i is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an if, while or for can include relational operators like <, <=, >, >=, == ("is equal to"), and != ("not equal to"); regular expression matches with the match operators ~ and !~; the logical operators ||, &&, and !; and of course parentheses for grouping.

The break statement causes an immediate exit from an enclosing while or for; the continue statement causes the next iteration to begin.

The statement next causes *awk* to skip immediately to the next record and begin scanning the patterns from the top. The statement exit causes the program to behave as if the end of the input had occurred.

Comments may be placed in *awk* programs: they begin with the character # and end with the end of the line, as in

```
print x, y # this is a comment
```

4. Design

The UNIX system already provides several programs that operate by passing input through a selection mechanism. *Grep*, the first and simplest, merely prints all lines which match a single specified pattern. *Egrep* provides more general patterns, i.e., regular expressions in full generality; *fgrep* searches for a set of keywords with a particularly fast algorithm. *Sed*[1] provides most of the editing facilities of the editor *ed*, applied to a stream of input. None of these programs provides numeric capabilities, logical relations, or variables.

Lex[3] provides general regular expression recognition capabilities, and, by serving as a C program generator, is essentially open-ended in its capabilities. The use of *lex*, however, requires a knowledge of C programming, and a *lex* program must be compiled and loaded before use, which discourages its use for one-shot applications.

Awk is an attempt to fill in another part of the matrix of possibilities. It provides general regular expression capabilities and an implicit input/output loop. But it also provides convenient numeric processing, variables, more general selection, and control flow in the actions. It does not require compilation or a knowledge of C. Finally, *awk* provides a convenient way to access fields within lines; it is unique in this respect.

Awk also tries to integrate strings and numbers completely, by treating all quantities as both string and numeric, deciding which representation is appropriate as late as possible. In most cases the user can simply ignore the differences.

Most of the effort in developing *awk* went into deciding what *awk* should or should not do (for instance, it doesn't do string substitution) and what the syntax should be (no explicit operator for concatenation) rather than on writing or debugging the code. We have tried to make the syntax powerful but easy to use and well adapted to scanning files. For example, the absence of declarations and implicit initializations, while probably a bad idea for a general-purpose programming language, is desirable in a language that is meant to be used for tiny programs that may even be composed on the command line.

In practice, *awk* usage seems to fall into two broad categories. One is what might be called "report generation" — processing an input to extract counts, sums, sub-totals, etc. This also includes the writing of trivial data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly balanced. The combination of textual and numeric processing is invaluable here.

A second area of use is as a data transformer, converting data from the form produced by one program into that expected by another. The simplest examples merely select fields, perhaps with rearrangements.

5. Implementation

The actual implementation of *awk* uses the language development tools available on the UNIX operating system. The grammar is specified with *yacc*;[4] the lexical analysis is done by *lex*; the regular expression recognizers are deterministic finite automata constructed directly from the expressions. An *awk* program is translated into a parse tree which is then directly executed by a simple interpreter.

Awk was designed for ease of use rather than processing speed; the delayed evaluation of variable types and the necessity to break input into fields makes high speed difficult to achieve in any case. Nonetheless, the program has not proven to be unworkably slow.

Table I below shows the execution (user + system) time on a PDP-11/70 of the UNIX programs *wc*, *grep*, *egrep*, *fgrep*, *sed*, *lex*, and *awk* on the following simple tasks:

1. count the number of lines.
2. print all lines containing "doug".
3. print all lines containing "doug", "ken" or "dmr".
4. print the third field of each line.
5. print the third and second fields of each line, in that order.
6. append all lines containing "doug", "ken", and "dmr" to files "jdoug", "jken", and "jdmr", respectively.
7. print each line prefixed by "line-number : ".
8. sum the fourth column of a table.

The program *wc* merely counts words, lines and characters in its input; we have already mentioned the others. In all cases the input was a file containing 10,000 lines as created by the command *ls −l*; each line has the form

```
−rw−rw−rw− 1 ava 123 Oct 15 17:05 xxx
```

The total length of this input is 452,960 characters. Times for *lex* do not include compile or load.

As might be expected, *awk* is not as fast as the specialized tools *wc*, *sed*, or the programs in the *grep* family, but is faster than the more general tool *lex*. In all cases, the tasks were about as easy to express as *awk* programs as programs in these other languages; tasks involving fields were considerably easier to express as *awk* programs. Some of the test programs are shown in *awk*, *sed* and *lex*.

References

1. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories (May 1975). Sixth Edition
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).
3. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).
4. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).

Program	Task 1	2	3	4	5	6	7	8
wc	8.6							
grep	11.7	13.1						
egrep	6.2	11.5	11.6					
fgrep	7.7	13.8	16.1					
sed	10.2	11.6	15.8	29.0	30.5	16.1		
lex	65.1	150.1	144.2	67.7	70.3	104.0	81.7	92.8
awk	15.0	25.6	29.9	33.3	38.9	46.4	71.4	31.1

Table I. Execution Times of Programs. (Times are in sec.)

The programs for some of these jobs are shown below. The *lex* programs are generally too long to show.

AWK:

1. `END {print NR}`

2. `/doug/`

3. `/ken|doug|dmr/`

4. `{print $3}`

5. `{print $3, $2}`

6.
```
/ken/     {print >"jken"}
/doug/    {print >"jdoug"}
/dmr/     {print >"jdmr"}
```

7. `{print NR ": " $0}`

8.
```
        {sum = sum + $4}
END {print sum}
```

SED:

1. `$=`

2. `/doug/p`

3.
```
/doug/p
/doug/d
/ken/p
/ken/d
/dmr/p
/dmr/d
```

4. `/[^ ]* [ ]*[^ ]* [ ]*\([^ ]*\) .*/s//\1/p`

5. `/[^ ]* [ ]*\([^ ]*\) [ ]*\([^ ]*\) .*/s//\2 \1/p`

6.
```
/ken/w jken
/doug/w jdoug
/dmr/w jdmr
```

LEX:

1.
```
%{
int i;
%}
%%
\n    i++;
.    ;
%%
yywrap() {
        printf("%d\n", i);
}
```

2.
```
%%
^.*doug.*$          printf("%s\n", yytext);
.    ;
\n   ;
```

Typing Documents on the UNIX System: Using the −ms Macros with Troff and Nroff

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This document describes a set of easy-to-use macros for preparing documents on the UNIX system. Documents may be produced on either the phototypesetter or a on a computer terminal, without changing the input.

The macros provide facilities for paragraphs, sections (optionally with automatic numbering), page titles, footnotes, equations, tables, two-column format, and cover pages for papers.

This memo includes, as an appendix, the text of the "Guide to Preparing Documents with −ms" which contains additional examples of features of −ms.

This manual is a revision of, and replaces, "Typing Documents on UNIX," dated November 22, 1974.

November 13, 1978

Typing Documents on the UNIX System: Using the −ms Macros with Troff and Nroff

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction. This memorandum describes a package of commands to produce papers using the *troff* and *nroff* formatting programs on the UNIX system. As with other *roff*-derived programs, text is prepared interspersed with formatting commands. However, this package, which itself is written in *troff* commands, provides higher-level commands than those provided with the basic *troff* program. The commands available in this package are listed in Appendix A.

Text. Type normally, except that instead of indenting for paragraphs, place a line reading ".PP" before each paragraph. This will produce indenting and extra space.

Alternatively, the command .LP that was used here will produce a left-aligned (block) paragraph. The paragraph spacing can be changed: see below under "Registers."

Beginning. For a document with a paper-type cover sheet, the input should start as follows:

```
[optional overall format .RP — see below]
.TL
Title of document (one or more lines)
.AU
Author(s) (may also be several lines)
.AI
Author's institution(s)
.AB
Abstract; to be placed on the cover sheet of a paper.
Line length is 5/6 of normal; use .ll here to change.
.AE  (abstract end)
text ... (begins with .PP, which see)
```

To omit some of the standard headings (e.g. no abstract, or no author's institution) just omit the corresponding fields and command lines. The word ABSTRACT can be suppressed by writing ".AB no" for ".AB". Several interspersed .AU and .AI lines can be used for multiple authors. The headings are not compulsory: beginning with a .PP command is perfectly OK and will just start printing an ordinary paragraph. ***Warning:*** You can't just begin a document with a line of text. Some −ms command must precede any text input. When in doubt, use .LP to get proper initialization, although any of the commands .PP, .LP, .TL, .SH, .NH is good enough. Figure 1 shows the legal arrangement of commands at the start of a document.

Cover Sheets and First Pages. The first line of a document signals the general format of the first page. In particular, if it is ".RP" a cover sheet with title and abstract is prepared. The default format is useful for scanning drafts.

In general −ms is arranged so that only one form of a document need be stored, containing all information; the first command gives the format, and unnecessary items for that format are ignored.

Warning: don't put extraneous material between the .TL and .AE commands. Processing of the titling items is special, and other data placed in them may not behave as you expect. Don't forget that some −ms command must precede any input text.

Page headings. The −ms macros, by default, will print a page heading containing a page number (if greater than 1). A default page footer is provided only in *nroff*, where the date is used. The user can make minor adjustments to the page headings/footings by redefining the strings LH, CH, and RH which are the left, center and right portions of the page headings, respectively; and the strings LF, CF, and RF, which are the left, center and right portions of the page footer. For more complex formats, the user can redefine the macros PT and BT, which are invoked respectively at the top and bottom of each page. The margins (taken from registers HM and FM for the top and bottom margin respectively) are normally 1 inch; the page header/footer are in the middle of that space. The user who redefines these macros should be careful not to change parameters such as point size or font without resetting them to default values.

Multi-column formats. If you place the command ".2C" in your document, the document will be printed in double column format beginning at that point. This feature is not too useful in computer terminal output, but is often desirable on the typesetter. The command ".1C" will go back to one-column format and also skip to a new page. The ".2C" command is actually a special case of the command

```
.MC [column width [gutter width]]
```

which makes multiple columns with the specified column and gutter width; as many columns as will fit across the page are used. Thus triple, quadruple, ... column pages can be printed. Whenever the number of columns is changed (except going from full width to some larger number of columns) a new page is started.

Headings. To produce a special heading, there are two commands. If you type

```
.NH
type section heading here
may be several lines
```

you will get automatically numbered section headings (1, 2, 3, ...), in boldface. For example,

```
.NH
Care and Feeding of Department Heads
```

produces

1. Care and Feeding of Department Heads

Alternatively,

```
.SH
Care and Feeding of Directors
```

will print the heading with no number added:

Care and Feeding of Directors

Every section heading, of either type, should be followed by a paragraph beginning with .PP or .LP, indicating the end of the heading. Headings may contain more than one line of text.

The .NH command also supports more complex numbering schemes. If a numerical argument is given, it is taken to be a "level" number and an appropriate sub-section number is generated. Larger level numbers indicate deeper sub-sections, as in this example:

```
.NH
Erie-Lackawanna
.NH 2
Morris and Essex Division
.NH 3
Gladstone Branch
.NH 3
Montclair Branch
.NH 2
Boonton Line
```

generates:

2. Erie-Lackawanna

2.1. Morris and Essex Division

2.1.1. Gladstone Branch

2.1.2. Montclair Branch

2.2. Boonton Line

An explicit ".NH 0" will reset the numbering of level 1 to one, as here:

```
.NH 0
Penn Central
```

1. Penn Central

Indented paragraphs. (Paragraphs with hanging numbers, e.g. references.) The sequence

```
.IP [1]
Text for first paragraph, typed
normally for as long as you would
like on as many lines as needed.
.IP [2]
Text for second paragraph, ...
```

produces

[1] Text for first paragraph, typed normally for as long as you would like on as many lines as needed.

[2] Text for second paragraph, ...

A series of indented paragraphs may be followed by an ordinary paragraph beginning with .PP or .LP, depending on whether you wish indenting or not. The command .LP was used here.

More sophisticated uses of .IP are also possible. If the label is omitted, for example, a plain block indent is produced.

```
.IP
This material will
just be turned into a
block indent suitable for quotations or
such matter.
.LP
```

will produce

> This material will just be turned into a block indent suitable for quotations or such matter.

If a non-standard amount of indenting is required, it may be specified after the label (in character positions) and will remain in effect until the next .PP or .LP. Thus, the general form of the .IP command contains two additional fields: the label and the indenting length. For example,

```
.IP first: 9
Notice the longer label, requiring larger
indenting for these paragraphs.
.IP second:
And so forth.
.LP
```

produces this:

first: Notice the longer label, requiring larger indenting for these paragraphs.

second: And so forth.

It is also possible to produce multiple nested indents; the command .RS indicates that the next .IP starts from the current indentation level. Each .RE will eat up one level of indenting so you should balance .RS and .RE commands. The .RS command should be thought of as "move right" and the .RE command as "move left". As an example

```
.IP 1.
Bell Laboratories
.RS
.IP 1.1
Murray Hill
.IP 1.2
Holmdel
.IP 1.3
Whippany
.RS
.IP 1.3.1
Madison
.RE
.IP 1.4
Chester
.RE
.LP
```

will result in

```
1.    Bell Laboratories
      1.1   Murray Hill
      1.2   Holmdel
      1.3   Whippany
            1.3.1 Madison
      1.4   Chester
```

All of these variations on .LP leave the right margin untouched. Sometimes, for purposes such as setting off a quotation, a paragraph indented on both right and left is required.

> A single paragraph like this is obtained by preceding it with .QP. More complicated material (several paragraphs) should be bracketed with .QS and .QE.

Emphasis. To get italics (on the typesetter) or underlining (on the terminal) say

```
.I
as much text as you want
can be typed here
.R
```

as was done for *these three words*. The .R command restores the normal (usually Roman) font. If only one word is to be italicized, it may be just given on the line with the .I command,

```
.I word
```

and in this case no .R is needed to restore the previous font. **Boldface** can be produced by

```
.B
Text to be set in boldface
goes here
.R
```

and also will be underlined on the terminal or line printer. As with .I, a single word can be placed in boldface by placing it on the same line as the .B command.

A few size changes can be specified similarly with the commands .LG (make larger), .SM (make smaller), and .NL (return to normal size). The size change is two points; the commands may be repeated for increased effect (here one .NL canceled two .SM commands).

If actual underlining as opposed to italicizing is required on the typesetter, the command

```
.UL word
```

will underline a word. There is no way to underline multiple words on the typesetter.

Footnotes. Material placed between lines with the commands .FS (footnote) and .FE (footnote end) will be collected, remembered, and finally placed at the bottom of the current page*. By default, footnotes are 11/12th the length of normal text, but this can be changed using the FL register (see below).

Displays and Tables. To prepare displays of lines, such as tables, in which the lines should not be re-arranged, enclose them in the commands .DS and .DE

```
.DS
table lines, like the
examples here, are placed
between .DS and .DE
.DE
```

By default, lines between .DS and .DE are indented and left-adjusted. You can also center lines, or retain the left margin. Lines bracketed by .DS C and .DE commands are centered (and not re-arranged); lines bracketed by .DS L and .DE are left-adjusted, not indented, and not re-arranged. A plain .DS is equivalent to .DS I, which indents and left-adjusts. Thus,

```
these lines were preceded
by .DS C and followed by
     a .DE command;
```

whereas

```
these lines were preceded
by .DS L and followed by
a .DE command.
```

Note that .DS C centers each line; there is a variant .DS B that makes the display into a left-adjusted block of text, and then centers that entire block. Normally a display is kept together, on one page. If you wish to have a long display which may be split across page boundaries, use .CD, .LD, or .ID in place of the commands .DS C, .DS L, or .DS I respectively. An extra argument to the .DS I or .DS command is taken as an amount to indent. Note: it is tempting to assume that .DS R will right adjust lines, but it doesn't work.

Boxing words or lines. To draw rectangular boxes around words the command

```
.BX word
```

will print $\boxed{\text{word}}$ as shown. The boxes will not be neat on a terminal, and this should not be used as a substitute for italics.

Longer pieces of text may be boxed by enclosing them with .B1 and .B2: .B1 text... .B2 as has been done here.

Keeping blocks together. If you wish to keep a table or other block of lines together on a page, there are "keep -

* Like this.

release'' commands. If a block of lines preceded by .KS and followed by .KE does not fit on the remainder of the current page, it will begin on a new page. Lines bracketed by .DS and .DE commands are automatically kept together this way. There is also a ''keep floating'' command: if the block to be kept together is preceded by .KF instead of .KS and does not fit on the current page, it will be moved down through the text until the top of the next page. Thus, no large blank space will be introduced in the document.

Nroff/Troff commands. Among the useful commands from the basic formatting programs are the following. They all work with both typesetter and computer terminal output:

```
.bp - begin new page.
.br - ''break'', stop running text
        from line to line.
.sp n - insert n blank lines.
.na - don't adjust right margins.
```

Date. By default, documents produced on computer terminals have the date at the bottom of each page; documents produced on the typesetter don't. To force the date, say ''.DA''. To force no date, say ''.ND''. To lie about the date, say ''.DA July 4, 1776'' which puts the specified date at the bottom of each page. The command

```
.ND May 8, 1945
```

in ".RP" format places the specified date on the cover sheet and nowhere else. Place this line before the title.

Signature line. You can obtain a signature line by placing the command .SG in the document. The authors' names will be output in place of the .SG line. An argument to .SG is used as a typing identification line, and placed after the signatures. The .SG command is ignored in released paper format.

Registers. Certain of the registers used by −ms can be altered to change default settings. They should be changed with .nr commands, as with

```
.nr PS 9
```

to make the default point size 9 point. If the effect is needed immediately, the normal *troff* command should be used in addition to changing the number register.

Register	Defines	Takes effect	Default
PS	point size	next para.	10
VS	line spacing	next para.	12 pts
LL	line length	next para.	6"
LT	title length	next para.	6"
PD	para. spacing	next para.	0.3 VS
PI	para. indent	next para.	5 ens
FL	footnote length	next FS	11/12 LL
CW	column width	next 2C	7/15 LL
GW	intercolumn gap	next 2C	1/15 LL
PO	page offset	next page	26/27"
HM	top margin	next page	1"
FM	bottom margin	next page	1"

You may also alter the strings LH, CH, and RH which are the left, center, and right headings respectively; and similarly LF, CF, and RF which are strings in the page footer. The page number on *output* is taken from register PN, to permit changing its output style. For more complicated headers and footers the macros PT and BT can be redefined, as explained earlier.

Accents. To simplify typing certain foreign words, strings representing common accent marks are defined. They precede the letter over which the mark is to appear. Here are the strings:

Input	Output	Input	Output
*'e	é	*~a	ã
*\`e	è	*Ce	ě
*:u	ü	*,c	c
*^e	ê		

Use. After your document is prepared and stored on a file, you can print it on a terminal with the command*

```
nroff −ms file
```

and you can print it on the typesetter with the command

```
troff −ms file
```

(many options are possible). In each case, if your document is stored in several files, just list all the filenames where we have used ''file''. If equations or tables are used, *eqn* and/or *tbl* must be invoked as preprocessors.

* If .2C was used, pipe the *nroff* output through *col;* make the first line of the input ''.pi /usr/bin/col.''

References and further study. If you have to do Greek or mathematics, see *eqn* [1] for equation setting. To aid *eqn* users, *−ms* provides definitions of .EQ and .EN which normally center the equation and set it off slightly. An argument on .EQ is taken to be an equation number and placed in the right margin near the equation. In addition, there are three special arguments to EQ: the letters C, I, and L indicate centered (default), indented, and left adjusted equations, respectively. If there is both a format argument and an equation number, give the format argument first, as in

.EQ L (1.3a)

for a left-adjusted equation numbered (1.3a).

Similarly, the macros .TS and .TE are defined to separate tables (see [2]) from text with a little space. A very long table with a heading may be broken across pages by beginning it with .TS H instead of .TS, and placing the line .TH in the table data after the heading. If the table has no heading repeated from page to page, just use the ordinary .TS and .TE macros.

To learn more about *troff* see [3] for a general introduction, and [4] for the full details (experts only). Information on related UNIX commands is in [5]. For jobs that do not seem well-adapted to −ms, consider other macro packages. It is often far easier to write a specific macro packages for such tasks as imitating particular journals than to try to adapt −ms.

Acknowledgment. Many thanks are due to Brian Kernighan for his help in the design and implementation of this package, and for his assistance in preparing this manual.

References

[1] B. W. Kernighan and L. L. Cherry, *Typesetting Mathematics — Users Guide (2nd edition)*, Bell Laboratories Computing Science Report no. 17.

[2] M. E. Lesk, *Tbl — A Program to Format Tables*, Bell Laboratories Computing Science Report no. 45.

[3] B. W. Kernighan, *A Troff Tutorial*, Bell Laboratories, 1976.

[4] J. F. Ossanna, *Nroff/Troff Reference Manual*, Bell Laboratories Computing Science Report no. 51.

[5] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Bell Laboratories, 1978.

Appendix A
List of Commands

1C	Return to single column format.
2C	Start double column format.
AB	Begin abstract.
AE	End abstract.
AI	Specify author's institution.
AU	Specify author.
B	Begin boldface.
DA	Provide the date on each page.
DE	End display.
DS	Start display (also CD, LD, ID).
EN	End equation.
EQ	Begin equation.
FE	End footnote.
FS	Begin footnote.
I	Begin italics.
IP	Begin indented paragraph.
KE	Release keep.
KF	Begin floating keep.
KS	Start keep.
LG	Increase type size.
LP	Left aligned block paragraph.
ND	Change or cancel date.
NH	Specify numbered heading.
NL	Return to normal type size.
PP	Begin paragraph.
R	Return to regular font (usually Roman).
RE	End one level of relative indenting.
RP	Use released paper format.
RS	Relative indent increased one level.
SG	Insert signature line.
SH	Specify section heading.
SM	Change to smaller type size.
TL	Specify title.
UL	Underline one word.

Register Names

The following register names are used by −ms internally. Independent use of these names in one's own macros may produce incorrect output. Note that no lower case letters are used in any −ms internal name.

Number registers used in −ms

:	DW	GW	HM	IQ	LL	NA	OJ	PO	T.	TV
#T	EF	H1	HT	IR	LT	NC	PD	PQ	TB	VS
1T	FL	H3	IK	KI	MM	NF	PF	PX	TD	YE
AV	FM	H4	IM	L1	MN	NS	PI	RO	TN	YY
CW	FP	H5	IP	LE	MO	OI	PN	ST	TQ	ZN

String registers used in −ms

'	A5	CB	DW	EZ	I	KF	MR	R1	RT	TL
`	AB	CC	DY	FA	I1	KQ	ND	R2	S0	TM
^	AE	CD	E1	FE	I2	KS	NH	R3	S1	TQ
-	AI	CF	E2	FJ	I3	LB	NL	R4	S2	TS
:	AU	CH	E3	FK	I4	LD	NP	R5	SG	TT
,	B	CM	E4	FN	I5	LG	OD	RC	SH	UL
1C	BG	CS	E5	FO	ID	LP	OK	RE	SM	WB
2C	BT	CT	EE	FQ	IE	ME	PP	RF	SN	WH
A1	C	D	EL	FS	IM	MF	PT	RH	SY	WT
A2	C1	DA	EM	FV	IP	MH	PY	RP	TA	XD
A3	C2	DE	EN	FY	IZ	MN	QF	RQ	TE	XF
A4	CA	DS	EQ	HO	KE	MO	R	RS	TH	XK

Order of Commands in Input

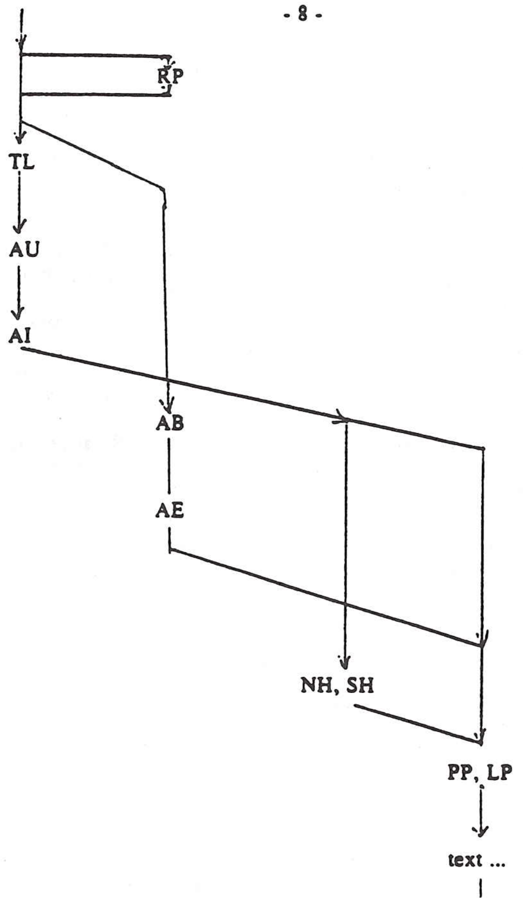


Figure 1

A Guide to Preparing Documents with −ms

M. E. Lesk

Bell Laboratories

August 1978

This guide gives some simple examples of document preparation on Bell Labs computers, emphasizing the use of the *−ms* macro package. It enormously abbreviates information in

1. *Typing Documents on UNIX and GCOS,* by M. E. Lesk;
2. *Typesetting Mathematics — User's Guide,* by B. W. Kernighan and L. L. Cherry; and
3. *Tbl — A Program to Format Tables,* by M. E. Lesk.

These memos are all included in the *UNIX Programmer's Manual, Volume 2.* The new user should also have *A Tutorial Introduction to the UNIX Text Editor,* by B. W. Kernighan.

For more detailed information, read *Advanced Editing on UNIX* and *A Troff Tutorial,* by B. W. Kernighan, and (for experts) *Nroff/Troff Reference Manual* by J. F. Ossanna. Information on related commands is found (for UNIX users) in *UNIX for Beginners* by B. W. Kernighan and the *UNIX Programmer's Manual* by K. Thompson and D. M. Ritchie.

Contents

Throughout the examples, input is shown in
```
this Helvetica sans serif font
```
while the resulting output is shown in
  this Times Roman font.

UNIX Document no. 1111

Commands for a TM

```
.TM 1978-5b3 99999 99999-11
.ND April 1, 1976
.TL
The Role of the Allen Wrench in Modern
Electronics
.AU "MH 2G-111" 2345
J. Q. Pencilpusher
.AU "MH 1K-222" 5432
X. Y. Hardwired
.AI
.MH
.OK
Tools
Design
.AB
This abstract should be short enough to
fit on a single page cover sheet.
It must attract the reader into sending for
the complete memorandum.
.AE
.CS 10 2 12 5 6 7
.NH
Introduction.
.PP
Now the first paragraph of actual text ...
...
Last line of text.
.SG MH-1234-JQP/XYH-unix
.NH
References ...
```

Commands not needed in a particular format are ignored.

Bell Laboratories   Cover Sheet for TM

This information is for employees of Bell Laboratories. (GEI 13.9-3)

Title- The Role of the Allen Wrench in Modern Electronics   Date- April 1, 1976

TM- 1978-5b3

Other Keywords- Tools
Design

Author	Location	Ext.	
J. Q. Pencilpusher	MH 2G-111	2345	Charging Case- 99999
X. Y. Hardwired	MH 1K-222	5432	Filing Case- 99999a

ABSTRACT

This abstract should be short enough to fit on a single page cover sheet. It must attract the reader into sending for the complete memorandum.

Pages Text 10	Other 2	Total 12
No. Figures 5	No. Tables 6	No. Refs. 7

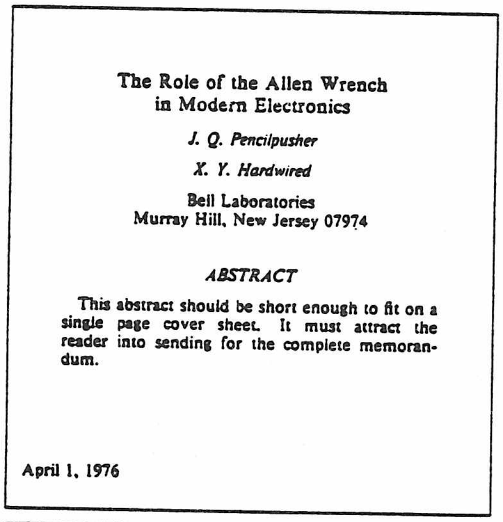
E-1932-U (6-73)   SEE REVERSE SIDE FOR DISTRIBUTION LIST

A Released Paper with Mathematics

```
.EQ
delim $$
.EN
.RP

... (as for a TM)

.CS 10 2 12 5 6 7
.NH
Introduction
.PP
The solution to the torque handle equation
.EQ (1)
sum from 0 to inf F ( x sub i ) = G ( x )
.EN
is found with the transformation $ x = rho over
theta $ where $ rho = G prime (x) $ and $theta$
is derived ...
```



The Role of the Allen Wrench
in Modern Electronics

J. Q. Pencilpusher

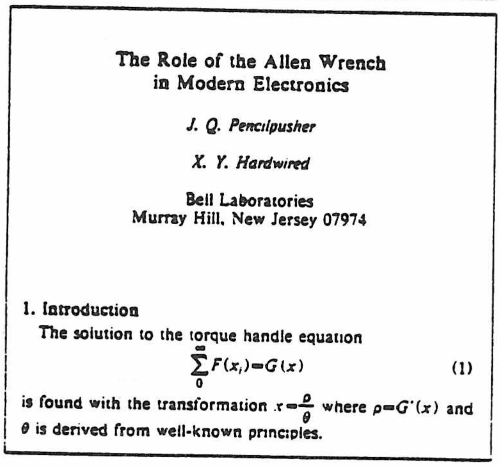
X. Y. Hardwired

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This abstract should be short enough to fit on a single page cover sheet. It must attract the reader into sending for the complete memorandum.

April 1, 1976



The Role of the Allen Wrench
in Modern Electronics

J. Q. Pencilpusher

X. Y. Hardwired

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

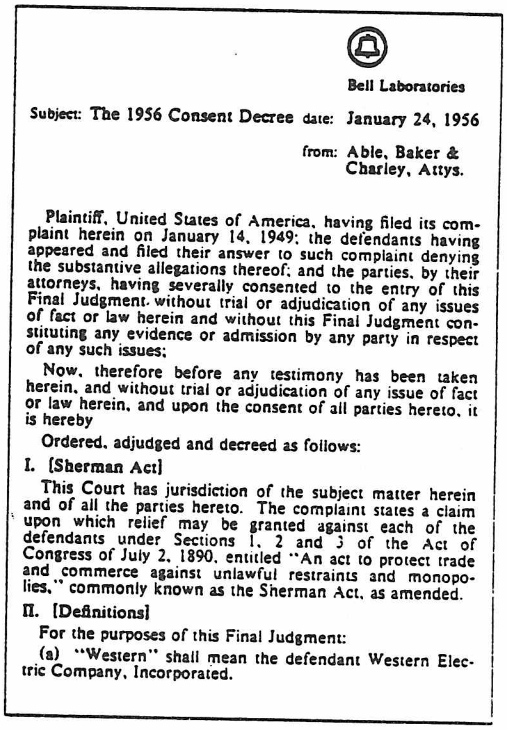
The solution to the torque handle equation

$$\sum_0^\infty F(x_i) = G(x) \qquad (1)$$

is found with the transformation $x = \frac{\rho}{\theta}$ where $\rho = G'(x)$ and θ is derived from well-known principles.

An Internal Memorandum

```
.IM
.ND January 24, 1956
.TL
The 1956 Consent Decree
.AU
Able, Baker &
Charley, Attys.
.PP
Plaintiff, United States of America, having filed
its complaint herein on January 14, 1949; the
defendants having appeared and filed their
answer to such complaint denying the
substantive allegations thereof; and the parties,
by their attorneys, ...
```



Bell Laboratories

Subject: The 1956 Consent Decree date: January 24, 1956

from: Able, Baker &
Charley, Attys.

Plaintiff, United States of America, having filed its complaint herein on January 14, 1949; the defendants having appeared and filed their answer to such complaint denying the substantive allegations thereof; and the parties, by their attorneys, having severally consented to the entry of this Final Judgment without trial or adjudication of any issues of fact or law herein and without this Final Judgment constituting any evidence or admission by any party in respect of any such issues;

Now, therefore before any testimony has been taken herein, and without trial or adjudication of any issue of fact or law herein, and upon the consent of all parties hereto, it is hereby

Ordered, adjudged and decreed as follows:

I. [Sherman Act]

This Court has jurisdiction of the subject matter herein and of all the parties hereto. The complaint states a claim upon which relief may be granted against each of the defendants under Sections 1, 2 and 3 of the Act of Congress of July 2, 1890, entitled "An act to protect trade and commerce against unlawful restraints and monopolies," commonly known as the Sherman Act, as amended.

II. [Definitions]

For the purposes of this Final Judgment:

(a) "Western" shall mean the defendant Western Electric Company, Incorporated.

Other formats possible (specify before .TL) are: .MR ("memo for record"), .MF ("memo for file"), .EG ("engineer's notes") and .TR (Computing Science Tech. Report).

Headings

```
.NH                          .SH
Introduction.                Appendix I
.PP                          .PP
text text text               text text text
```

1. Introduction

text text text

Appendix I

text text text

A Simple List

```
.IP 1.
J. Pencilpusher and X. Hardwired,
.I
A New Kind of Set Screw,
.R
Proc. IEEE
.B 75
(1976), 23-255.
.IP 2.
H. Nails and R. Irons,
.I
Fasteners for Printed Circuit Boards,
.R
Proc. ASME
.B 23
(1974), 23-24.
.LP (terminates list)
```

1. J. Pencilpusher and X. Hardwired, *A New Kind of Set Screw,* Proc. IEEE **75** (1976), 23-255.
2. H. Nails and R. Irons, *Fasteners for Printed Circuit Boards,* Proc. ASME **23** (1974), 23-24.

Displays

```
text text text text text text
.DS
and now
for something
completely different
.DE
text text text text text text
```

hoboken harrison newark roseville avenue grove street east orange brick church orange highland avenue mountain station south orange maplewood millburn short hills summit new providence

    and now
    for something
    completely different

murray hill berkeley heights gillette stirling millington lyons basking ridge bernardsville far hills peapack gladstone

Options: .DS L: left-adjust; .DS C: line-by-line center; .DS B: make block, then center.

Footnotes

```
Among the most important occupants
of the workbench are the long-nosed pliers.
Without these basic tools*
.FS
* As first shown by Tiger & Leopard
(1975).
.FE
few assemblies could be completed. They may
lack the popular appeal of the sledgehammer
```

Among the most important occupants of the workbench are the long-nosed pliers. Without these basic tools[*] few assemblies could be completed. They may lack the popular appeal of the sledgehammer

[*] As first shown by Tiger & Leopard (1975).

Multiple Indents

```
This is ordinary text to point out
the margins of the page.
.IP 1.
First level item
.RS
.IP a)
Second level.
.IP b)
Continued here with another second
level item, but somewhat longer.
.RE
.IP 2.
Return to previous value of the
indenting at this point.
.IP 3.
Another
line.
```

This is ordinary text to point out the margins of the page.

1. First level item
   a) Second level.
   b) Continued here with another second level item, but somewhat longer.
2. Return to previous value of the indenting at this point.
3. Another line.

Keeps

Lines bracketed by the following commands are kept together, and will appear entirely on one page:

.KS	not moved	.KF	may float
.KE	through text	.KE	in text

Double Column

```
.TL
The Declaration of Independence
.2C
.PP
When in the course of human events, it becomes
necessary for one people to dissolve the
political bonds which have connected them with
another, and to assume among the powers of the
earth the separate and equal station to which
the laws of Nature and of Nature's God entitle
them, a decent respect to the opinions of
```

The Declaration of Independence

When in the course of human events, it becomes necessary for one people to dissolve the political bonds which have connected them with another, and to assume among the powers of the earth the separate and equal station to which the laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident, that all men are created equal, that they are endowed by their creator with certain unalienable rights, that among these are life, liberty, and the pursuit of happiness. That to secure these rights, governments are instituted among men,

Equations

```
A displayed equation is marked
with an equation number at the right margin
by adding an argument to the EQ line:
.EQ (1.3)
x sup 2 over a sup 2 ~=~ sqrt {p z sup 2 +qz+r}
.EN
```

A displayed equation is marked with an equation number at the right margin by adding an argument to the EQ line:

$$\frac{x^2}{a^2} = \sqrt{pz^2+qz+r} \qquad (1.3)$$

```
.EQ I (2.2a)
bold V bar sub nu~=~left [ pile {a above b above
c } right ] + left [ matrix { col { A(11) above .
above . } col { . above . above .} col {. above .
above A(33) }} right ] cdot left [ pile { alpha
above beta above  gamma } right ]
.EN
```

$$\bar{\mathbf{V}}_\nu = \begin{bmatrix} a \\ b \\ c \end{bmatrix} + \begin{bmatrix} A(11) & . & . \\ . & . & . \\ . & . & A(33) \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \qquad (2.2a)$$

```
.EQ L
F hat ( chi ) ~ mark = ~ | del V | sup 2
.EN
.EQ L
lineup =~ {left ( {partial V} over {partial x} right )
} sup 2 + { left ( {partial V} over {partial y} right
) } sup 2 ~~~~~~ lambda -> inf
.EN
```

$$\hat{F}(\chi) = |\nabla V|^2$$

$$= \left(\frac{\partial V}{\partial x}\right)^2 + \left(\frac{\partial V}{\partial y}\right)^2 \qquad \lambda \to \infty$$

$ a dot $, $ b dotdot$, $ xi tilde times y vec$:

$\dot{a}$, $\ddot{b}$, $\tilde{\xi} \times \vec{y}$. (with delim $$ on, see panel 3).

See also the equations in the second table, panel 8.

Some Registers You Can Change

Line length .nr LL 7i	Paragraph spacing .nr PD 0
Title length .nr LT 7i	Page offset .nr PO 0.5i
Point size .nr PS 9	Page heading .ds CH Appendix (center) .ds RH 7-25-76 (right) .ds LH Private (left)
Vertical spacing .nr VS 11	
Column width .nr CW 3i	
Intercolumn spacing .nr GW .5i	Page footer .ds CF Draft .ds LF similar .ds RF
Margins — head and foot .nr HM .75i .nr FM .75i	Page numbers .nr % 3
Paragraph indent .nr PI 2n	

Tables

```
.TS             (Ⓣ indicates a tab)
allbox;
c s s
c c c
n n n.
AT&T Common Stock
YearⓉPriceⓉDividend
1971Ⓣ41-54Ⓣ$2.60
2Ⓣ41-54Ⓣ2.70
3Ⓣ46-55Ⓣ2.87
4Ⓣ40-53Ⓣ3.24
5Ⓣ45-52Ⓣ3.40
6Ⓣ51-59Ⓣ.95*
.TE
* (first quarter only)
```

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

The meanings of the key-letters describing the alignment of each entry are:

c	center	n	numerical
r	right-adjust	a	subcolumn
l	left-adjust	s	spanned

The global table options are center, expand, box, doublebox, allbox, tab (x) and linesize (n).

```
.TS             (with delim $$ on, see panel 3)
doublebox, center;
c c
l l.
NameⓉDefinition
..sp
GammaⓉ$GAMMA (z) = int sub 0 sup inf \
  t sup {z-1} e sup -t dt$
SineⓉ$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
ErrorⓉ$ roman erf (z) = 2 over sqrt pi \
 int sub 0 sup z e sup {-t sup 2} dt$
BesselⓉ$ J sub 0 (z) = 1 over pi \
 int sub 0 sup pi cos ( z sin theta ) d theta $
ZetaⓉ$ zeta (s) = \
 sum from k=1 to inf k sup -s ~~( Re~s > 1)$
.TE
```

Name	Definition
Gamma	$\Gamma(z)=\int_0^\infty t^{z-1}e^{-t}dt$
Sine	$\sin(x)=\frac{1}{2i}(e^{ix}-e^{-ix})$
Error	$\mathrm{erf}(z)=\frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$
Bessel	$J_0(z)=\frac{1}{\pi}\int_0^\pi \cos(z\sin\theta)d\theta$
Zeta	$\zeta(s)=\sum_{k=1}^\infty k^{-s} \quad (\mathrm{Re}\ s>1)$

Usage

```
      Documents with just text:
troff -ms files
      With equations only:
eqn files | troff -ms
      With tables only:
tbl files | troff -ms
      With both tables and equations:
tbl files|eqn|troff -ms
```

The above generates STARE output on GCOS: replace −st with −ph for typesetter output.

A Revised Version of –ms

Bill Tuthill

Computing Services
University of California
Berkeley, CA 94720

The –ms macros have been slightly revised and rearranged. Because of the rearrangement, the new macros can be read by the computer in about half the time required by the previous version of –ms. This means that output will begin to appear between ten seconds and several minutes more quickly, depending on the system load. On long files, however, the savings in total time are not substantial. The old version of –ms is still available as –mos.

Several bugs in –ms have been fixed, including a bad problem with the .1C macro, minor difficulties with boxed text, a break induced by .EQ before initialization, the failure to set tab stops in displays, and several bothersome errors in the **refer** macros. Macros used only at Bell Laboratories have been removed. There are a few extensions to previous –ms macros, and a number of new macros, but all the documented –ms macros still work exactly as they did before, and have the same names as before. Output produced with –ms should look like output produced with –mos.

One important new feature is automatically numbered footnotes. Footnote numbers are printed by means of a pre-defined string (**), which you invoke separately from .FS and .FE. Each time it is used, this string increases the footnote number by one, whether or not you use .FS and .FE in your text. Footnote numbers will be superscripted on the phototypesetter and on daisy-wheel terminals, but on low-resolution devices (such as the lpr and a crt), they will be bracketed. If you use ** to indicate numbered footnotes, then the .FS macro will automatically include the footnote number at the bottom of the page. This footnote, for example, was produced as follows:[1]

```
This footnote, for example, was produced as follows:\**
.FS
     ...
.FE
```

If you are using ** to number footnotes, but want a particular footnote to be marked with an asterisk or a dagger, then give that mark as the first argument to .FS: †

```
then give that mark as the first argument to .FS: \(dg
.FS  \(dg
     ...
.FE
```

Footnote numbering will be temporarily suspended, because the ** string is not used. Instead of a dagger, you could use an asterisk * or double dagger ‡, represented as \\(dd.

Another new feature is a macro for printing theses according to Berkeley standards. This macro is called .TM, which stands for thesis mode. (It is much like the .th macro in –me.) It will put page numbers in the upper right-hand corner; number the first page; suppress the date; and doublespace everything except quotes, displays, and keeps. Use it at the top of each file making up your thesis.

[1] If you never use the "\**" string, no footnote numbers will appear anywhere in the text, including down here. The output footnotes will look exactly like footnotes produced with –mos.

† In the footnote, the dagger will appear where the footnote number would otherwise appear, as on the left.

Calling .TM defines the .CT macro for chapter titles, which skips to a new page and moves the pagenumber to the center footer. The .P1 (P one) macro can be used even without thesis mode to print the header on page 1, which is suppressed except in thesis mode. If you want roman numeral page numbering, use an ".af PN i" request.

There is a new macro especially for bibliography entries, called .XP, which stands for exdented paragraph. It will exdent the first line of the paragraph by \n(PI units, usually 5n (the same as the indent for the first line of a .PP). Most bibliographies are printed this way. Here are some examples of exdented paragraphs:

Lumley, Lyle S., *Sex in Crustaceans: Shell Fish Habits,* Harbinger Press, Tampa Bay and San Diego, October 1979. 243 pages. The pioneering work in this field.

Leffadinger, Harry A., "Mollusk Mating Season: 52 Weeks, or All Year?" in *Acta Biologica,* vol. 42, no. 11, November 1980. A provocative thesis, but the conclusions are wrong.

Of course, you will have to take care of italicizing the book title and journal, and quoting the title of the journal article. Indentation or exdentation can be changed by setting the value of number register PI.

If you need to produce endnotes rather than footnotes, put the references in a file of their own. This is similar to what you would do if you were typing the paper on a conventional typewriter. Note that you can use automatic footnote numbering without actually having .FS and .FE pairs in your text. If you place footnotes in a separate file, you can use .IP macros with ** as a hanging tag; this will give you numbers at the left-hand margin. With some styles of endnotes, you would want to use .PP rather then .IP macros, and specify ** before the reference begins.

There are four new macros to help produce a table of contents. Table of contents entries must be enclosed in .XS and .XE pairs, with optional .XA macros for additional entries; arguments to .XS and .XA specify the page number, to be printed at the right. A final .PX macro prints out the table of contents. Here is a sample of typical input and output text:

```
.XS  ii
Introduction
.XA  1
Chapter 1: Review of the Literature
.XA  23
Chapter 2: Experimental Evidence
.XE
.PX
```

Table of Contents

The .XS and .XE pairs may also be used in the text, after a section header for instance, in which case page numbers are supplied automatically. However, most documents that require a table of contents are too long to produce in one run, which is necessary if this method is to work. It is recommended that you do a table of contents after finishing your document. To print out the table of contents, use the .PX macro; if you forget it, nothing will happen.

As an aid in producing text that will format correctly with both **nroff** and **troff**, there are some new string definitions that define quotation marks and dashes for each of these two formatting programs. The *– string will yield two hyphens in **nroff**, but in **troff** it will produce an em dash— like this one. The *Q and *U strings will produce " and " in **troff**, but " in **nroff**. (In typesetting, the double quote is traditionally considered bad form.)

There are now a large number of optional foreign accent marks defined by the –ms macros. All the accent marks available in –mos are present, and they all work just as they always did. However, there are better definitions available by placing .AM at the beginning of your document. Unlike the –mos accent marks, the accent strings should come *after* the letter being accented. Here is a list of the diacritical marks, with examples of what they look like.

name of accent	input	output
acute accent	e*´	é
grave accent	e*`	è
circumflex	o*^	ô
cedilla	c*,	ç
tilde	n*~	ñ
question	*?	¿
exclamation	*!	¡
umlaut	u*:	ü
digraph s	*8	β
haček	c*v	č
macron	a*_	ā
underdot	s*.	ṣ
o-slash	o*/	ø
angstrom	a*o	å
yogh	kni*3t	kniȝt
Thorn	*(Th	Þ
thorn	*(th	þ
Eth	*(D-	Đ
eth	*(d-	ð
hooked o	*q	ǫ
ae ligature	*(ae	æ
AE ligature	*(Ae	Æ
oe ligature	*(oe	œ
OE ligature	*(Oe	Œ

If you want to use these new diacritical marks, don't forget the .AM at the top of your file. Without it, some will not print at all, and others will be placed on the wrong letter.

It is also possible to produce custom headers and footers that are different on even and odd pages. The .OH and .EH macros define odd and even headers, while .OF and .EF define odd and even footers. Arguments to these four macros are specified as with .tl. This document was produced with:

```
.OH '\fIThe -mx Macros''Page %\fP'
.EH '\fIPage %''The -mx Macros\fP'
```

Note that it would be a error to have an apostrophe in the header text; if you need one, you will have to use a different delimiter around the left, center, and right portions of the title. You can use any character as a delimiter, provided it doesn't appear elsewhere in the argument to .OH, .EH, .OF, or EF.

The –ms macros work in conjunction with the **tbl**, **eqn**, and **refer** preprocessors. Macros to deal with these items are read in only as needed, as are the thesis macros (.TM), the special accent mark definitions (.AM), table of contents macros (.XS and .XE), and macros to format the optional cover page. The code for the –ms package lives in /usr/lib/tmac/tmac.s, and sourced files reside in the directory /usr/ucb/lib/ms.

WRITING PAPERS WITH NROFF USING −ME

Eric P. Allman

Electronics Research Laboratory
University of California, Berkeley
Berkeley, California 94720

This document describes the text processing facilities available on the UNIX† operating system via NROFF† and the −me macro package. It is assumed that the reader already is generally familiar with the UNIX operating system and a text editor such as ex. This is intended to be a casual introduction, and as such not all material is covered. In particular, many variations and additional features of the −me macro package are not explained. For a complete discussion of this and other issues, see *The −me Reference Manual* and *The NROFF/TROFF Reference Manual.*

NROFF, a computer program that runs on the UNIX operating system, reads an input file prepared by the user and outputs a formatted paper suitable for publication or framing. The input consists of *text*, or words to be printed, and *requests*, which give instructions to the NROFF program telling how to format the printed copy.

Section 1 describes the basics of text processing. Section 2 describes the basic requests. Section 3 introduces displays. Annotations, such as footnotes, are handled in section 4. The more complex requests which are not discussed in section 2 are covered in section 5. Finally, section 6 discusses things you will need to know if you want to typeset documents. If you are a novice, you probably won't want to read beyond section 4 until you have tried some of the basic features out.

When you have your raw text ready, call the NROFF formatter by typing as a request to the UNIX shell:

nroff −me −T*type files*

where *type* describes the type of terminal you are outputting to. Common values are **dtc** for a DTC 300s (daisy-wheel type) printer and **lpr** for the line printer. If the **−T** flag is omitted, a "lowest common denominator" terminal is assumed; this is good for previewing output on most terminals. A complete description of options to the NROFF command can be found in *The NROFF/TROFF Reference Manual.*

The word *argument* is used in this manual to mean a word or number which appears on the same line as a request which modifies the meaning of that request. For example, the request

```
.sp
```

spaces one line, but

```
.sp 4
```

spaces four lines. The number 4 is an *argument* to the **.sp** request which says to space four lines instead of one. Arguments are separated from the request and from each other by spaces.

†UNIX, NROFF, and TROFF are Trademarks of Bell Laboratories

1. Basics of Text Processing

The primary function of NROFF is to *collect* words from input lines, *fill* output lines with those words, *justify* the right hand margin by inserting extra spaces in the line, and output the result. For example, the input:

```
Now is the time
for all good men
to come to the aid
of their party.
Four score and seven
years ago,...
```

will be read, packed onto output lines, and justified to produce:

```
Now is the time for all good men to come to the aid of their party. Four score and
seven years ago,...
```

Sometimes you may want to start a new output line even though the line you are on is not yet full; for example, at the end of a paragraph. To do this you can cause a *break*, which starts a new output line. Some requests cause a break automatically, as do blank input lines and input lines beginning with a space.

Not all input lines are text to be formatted. Some of the input lines are *requests* which describe how to format the text. Requests always have a period or an apostrophe ("′") as the first character of the input line.

The text formatter also does more complex things, such as automatically numbering pages, skipping over page folds, putting footnotes in the correct place, and so forth.

I can offer you a few hints for preparing text for input to NROFF. First, keep the input lines short. Short input lines are easier to edit, and NROFF will pack words onto longer lines for you anyhow. In keeping with this, it is helpful to begin a new line after every period, comma, or phrase, since common corrections are to add or delete sentences or phrases. Second, do not put spaces at the end of lines, since this can sometimes confuse the NROFF processor. Third, do not hyphenate words at the end of lines (except words that should have hyphens in them, such as "mother-in-law"); NROFF is smart enough to hyphenate words for you as needed, but is not smart enough to take hyphens out and join a word back together. Also, words such as "mother-in-law" should not be broken over a line, since then you will get a space where not wanted, such as "mother- in-law".

2. Basic Requests

2.1. Paragraphs

Paragraphs are begun by using the **.pp** request. For example, the input:

```
.pp
Now is the time for all good men
to come to the aid of their party.
Four score and seven years ago,...
```

produces a blank line followed by an indented first line. The result is:

```
        Now is the time for all good men to come to the aid of their party. Four
score and seven years ago,...
```

Notice that the sentences of the paragraphs *must not* begin with a space, since blank lines and lines begining with spaces cause a break. For example, if I had typed:

```
.pp
Now is the time for all good men
    to come to the aid of their party.
Four score and seven years ago,...
```

The output would be:

```
     Now is the time for all good men
    to come to the aid of their party. Four score and seven years ago,...
```

A new line begins after the word "men" because the second line began with a space character.

There are many fancier types of paragraphs, which will be described later.

2.2. Headers and Footers

Arbitrary headers and footers can be put at the top and bottom of every page. Two requests of the form **.he** *title* and **.fo** *title* define the titles to put at the head and the foot of every page, respectively. The titles are called *three-part* titles, that is, there is a left-justified part, a centered part, and a right-justified part. To separate these three parts the first character of *title* (whatever it may be) is used as a delimiter. Any character may be used, but backslash and double quote marks should be avoided. The percent sign is replaced by the current page number whenever found in the title. For example, the input:

```
.he ''%''
.fo 'Jane Jones''My Book'
```

results in the page number centered at the top of each page, "Jane Jones" in the lower left corner, and "My Book" in the lower right corner.

2.3. Double Spacing

NROFF will double space output text automatically if you use the request **.ls 2**, as

is done in this section. You can revert to single spaced mode by typing **.ls 1**.

2.4. Page Layout

A number of requests allow you to change the way the printed copy looks, sometimes called the *layout* of the output page. Most of these requests adjust the placing of "white space" (blank lines or spaces). In these explanations, characters in italics should be replaced with values you wish to use; bold characters represent characters which should actually be typed.

The **.bp** request starts a new page.

The request **.sp** *N* leaves *N* lines of blank space. *N* can be omitted (meaning skip a single line) or can be of the form *N***i** (for *N* inches) or *N***c** (for *N* centimeters). For example, the input:

```
.sp 1.5i
My thoughts on the subject
.sp
```

leaves one and a half inches of space, followed by the line "My thoughts on the subject", followed by a single blank line.

The **.in** +*N* request changes the amount of white space on the left of the page (the *indent*). The argument *N* can be of the form **+***N* (meaning leave *N* spaces more than you are already leaving), **−***N* (meaning leave less than you do now), or just *N* (meaning leave exactly *N* spaces). *N* can be of the form *N***i** or *N***c** also. For example, the input:

```
initial text
.in 5
some text
.in +1i
more text
.in −2c
final text
```

produces "some text" indented exactly five spaces from the left margin, "more text" indented five spaces plus one inch from the left margin (fifteen spaces on a pica typewriter), and "final text" indented five spaces plus one inch minus two centimeters from the margin. That is, the output is:

```
initial text
          some text
                         more text
             final text
```

The **.ti** +*N* (temporary indent) request is used like **.in** +*N* when the indent should apply to one line only, after which it should revert to the previous indent. For example, the input:

```
.in 1i
.ti 0
Ware, James R. The Best of Confucius,
Halcyon House, 1950.
An excellent book containing translations of
most of Confucius' most delightful sayings.
A definite must for anyone interested in the early foundations
of Chinese philosophy.
```

produces:

```
Ware, James R. The Best of Confucius, Halcyon House, 1950. An excellent book con-
               taining translations of most of Confucius' most delightful sayings. A
               definite must for anyone interested in the early foundations of Chinese
               philosophy.
```

Text lines can be centered by using the **.ce** request. The line after the **.ce** is centered (horizontally) on the page. To center more than one line, use **.ce** *N* (where *N* is the number of lines to center), followed by the *N* lines. If you want to center many lines but don't want to count them, type:

```
.ce 1000
lines to center
.ce 0
```

The **.ce 0** request tells NROFF to center zero more lines, in other words, stop centering.

All of these requests cause a break; that is, they always start a new line. If you want to start a new line without performing any other action, use **.br**.

2.5. Underlining

Text can be underlined using the **.ul** request. The **.ul** request causes the next input line to be underlined when output. You can underline multiple lines by stating a count of *input* lines to underline, followed by those lines (as with the **.ce** request). For example, the input:

```
.ul 2
Notice that these two input lines
are underlined.
```

will underline those eight words in NROFF. (In TROFF they will be set in italics.)

3. Displays

Displays are sections of text to be set off from the body of the paper. Major quotes, tables, and figures are types of displays, as are all the examples used in this document. All displays except centered blocks are output single spaced.

3.1. Major Quotes

Major quotes are quotes which are several lines long, and hence are set in from the rest of the text without quote marks around them. These can be generated using the commmands **.(q** and **.)q** to surround the quote. For example, the input:

```
As Weizenbaum points out:
.(q
It is said that to explain is to explain away.
This maxim is nowhere so well fulfilled
as in the areas of computer programming,...
.)q
```

generates as output:

As Weizenbaum points out:

> It is said that to explain is to explain away. This maxim is nowhere so well fulfilled as in the areas of computer programming,...

3.2. Lists

A *list* is an indented, single spaced, unfilled display. Lists should be used when the material to be printed should not be filled and justified like normal text, such as columns of figures or the examples used in this paper. Lists are surrounded by the requests **.(l** and **.)l**. For example, type:

```
Alternatives to avoid deadlock are:
.(l
Lock in a specified order
Detect deadlock and back out one process
Lock all resources needed before proceeding
.)l
```

will produce:

Alternatives to avoid deadlock are:

```
Lock in a specified order
Detect deadlock and back out one process
Lock all resources needed before proceeding
```

3.3. Keeps

A *keep* is a display of lines which are kept on a single page if possible. An example of where you would use a keep might be a diagram. Keeps differ from lists in that lists may be broken over a page boundary whereas keeps will not.

Blocks are the basic kind of keep. They begin with the request **.(b** and end with the request **.)b**. If there is not room on the current page for everything in the block, a new page is begun. This has the unpleasant effect of leaving blank space at the bottom of the page. When this is not appropriate, you can use the alternative, called *floating keeps.*

Floating keeps move relative to the text. Hence, they are good for things which will be referred to by name, such as "See figure 3". A floating keep will appear at the bottom of the current page if it will fit; otherwise, it will appear at the top of the next page. Floating keeps begin with the line **.(z** and end with the line **.)z**. For an example of a floating keep, see figure 1. The **.hl** request is used to draw a horizontal line so that the figure stands out from the text.

3.4. Fancier Displays

Keeps and lists are normally collected in *nofill* mode, so that they are good for tables and such. If you want a display in fill mode (for text), type **.(l F** (Throughout this section, comments applied to **.(l** also apply to **.(b** and **.(z**). This kind of display will be indented from both margins. For example, the input:

```
.(l F
And now boys and girls,
a newer, bigger, better toy than ever before!
Be the first on your block to have your own computer!
Yes kids, you too can have one of these modern
data processing devices.
You too can produce beautifully formatted papers
without even batting an eye!
.)l
```

will be output as:

> And now boys and girls, a newer, bigger, better toy than ever before! Be the first on your block to have your own computer! Yes kids, you too can have one of these modern data processing devices. You too can produce beautifully formatted papers without even batting an eye!

Lists and blocks are also normally indented (floating keeps are normally left justified). To get a left-justified list, type **.(l L**. To get a list centered line-for-line, type **.(l C**. For example, to get a filled, left justified list, enter:

```
.(l L F
text of block
.)l
```

The input:

```
.(l
first line of unfilled display
more lines
.)l
```

produces the indented text:

```
.(z
.hl
Text of keep to be floated.
.sp
.ce
Figure 1.  Example of a Floating Keep.
.hl
.)z
```

Figure 1. Example of a Floating Keep.

```
    first line of unfilled display
    more lines
```

Typing the character **L** after the **.(l** request produces the left justified result:

```
first line of unfilled display
more lines
```

Using **C** instead of **L** produces the line-at-a-time centered output:

```
                    first line of unfilled display
                              more lines
```

Sometimes it may be that you want to center several lines as a group, rather than centering them one line at a time. To do this use centered blocks, which are surrounded by the requests **.(c** and **.)c**. All the lines are centered as a unit, such that the longest line is centered and the rest are lined up around that line. Notice that lines do not move relative to each other using centered blocks, whereas they do using the **C** argument to keeps.

Centered blocks are *not* keeps, and may be used in conjunction with keeps. For example, to center a group of lines as a unit and keep them on one page, use:

```
    .(b L
    .(c
    first line of unfilled display
    more lines
    .)c
    .)b
```

to produce:

```
                    first line of unfilled display
                    more lines
```

If the block requests (**.(b** and **.)b**) had been omitted the result would have been the same, but with no guarantee that the lines of the centered block would have all been on one page. Note the use of the **L** argument to **.(b**; this causes the centered block to center within the entire line rather than within the line minus the indent. Also, the center requests must be nested *inside* the keep requests.

4. Annotations

There are a number of requests to save text for later printing. *Footnotes* are printed at the bottom of the current page. *Delayed text* is intended to be a variant form of footnote; the text is printed only when explicitly called for, such as at the end of each chapter. *Indexes* are a type of delayed text having a tag (usually the page number) attached to each entry after a row of dots. Indexes are also saved until called for explicitly.

4.1. Footnotes

Footnotes begin with the request **.(f** and end with the request **.)f**. The current footnote number is maintained automatically, and can be used by typing **, to produce a footnote number[1]. The number is automatically incremented after every footnote. For example, the input:

[1]Like this.

```
.(q
A man who is not upright
and at the same time is presumptuous;
one who is not diligent and at the same time is ignorant;
one who is untruthful and at the same time is incompetent;
such men I do not count among acquaintances.\**
.(f
\**James R. Ware,
.ul
The Best of Confucius,
Halcyon House, 1950.
Page 77.
.)f
.)q
```

generates the result:

> A man who is not upright and at the same time is presumptuous; one who is not diligent and at the same time is ignorant; one who is untruthful and at the same time is incompetent; such men I do not count among acquaintances.[2]

It is important that the footnote appears *inside* the quote, so that you can be sure that the footnote will appear on the same page as the quote.

4.2. Delayed Text

Delayed text is very similar to a footnote except that it is printed when called for explicitly. This allows a list of references to appear (for example) at the end of each chapter, as is the convention in some disciplines. Use *# on delayed text instead of ** as on footnotes.

If you are using delayed text as your standard reference mechanism, you can still use footnotes, except that you may want to reference them with special characters[*] rather than numbers.

4.3. Indexes

An "index" (actually more like a table of contents, since the entries are not sorted alphabetically) resembles delayed text, in that it is saved until called for. However, each entry has the page number (or some other tag) appended to the last line of the index entry after a row of dots.

Index entries begin with the request **.(x** and end with **.)x**. The **.)x** request may have a argument, which is the value to print as the "page number". It defaults to the current page number. If the page number given is an underscore ("_") no page number or line of dots is printed at all. To get the line of dots without a page number, type **.)x ""**, which specifies an explicitly null page number.

The **.xp** request prints the index.

For example, the input:

[2]James R. Ware, *The Best of Confucius*, Halcyon House, 1950. Page 77.

[*]Such as an asterisk.

```
.(x
Sealing wax
.)x
.(x
Cabbages and kings
.)x _
.(x
Why the sea is boiling hot
.)x 2.5a
.(x
Whether pigs have wings
.)x ""
.(x
This is a terribly long index entry, such as might be used
for a list of illustrations, tables, or figures; I expect it to
take at least two lines.
.)x
.xp
```

generates:

```
Sealing wax ........................................................................ 9
Cabbages and kings
Why the sea is boiling hot ....................................................... 2.5a
Whether pigs have wings ..........................................................
This is a terribly long index entry, such as might be used for a list of illustra-
tions, tables, or figures; I expect it to take at least two lines. ............. 9
```

The **.(x** request may have a single character argument, specifying the "name" of the index; the normal index is **x**. Thus, several "indicies" may be maintained simultaneously (such as a list of tables, table of contents, etc.).

Notice that the index must be printed at the *end* of the paper, rather than at the beginning where it will probably appear (as a table of contents); the pages may have to be physically rearranged after printing.

5. Fancier Features

A large number of fancier requests exist, notably requests to provide other sorts of paragraphs, numbered sections of the form **1.2.3** (such as used in this document), and multicolumn output.

5.1. More Paragraphs

Paragraphs generally start with a blank line and with the first line indented. It is possible to get left-justified block-style paragraphs by using **.lp** instead of **.pp**, as demonstrated by the next paragraph.

Sometimes you want to use paragraphs that have the *body* indented, and the first line exdented (opposite of indented) with a label. This can be done with the **.ip** request. A word specified on the same line as **.ip** is printed in the margin, and the body is lined up at a prespecified position (normally five spaces). For example, the input:

```
.ip one
This is the first paragraph.
Notice how the first line
of the resulting paragraph lines up
with the other lines in the paragraph.
.ip two
And here we are at the second paragraph already.
You may notice that the argument to .ip
appears
in the margin.
.lp
We can continue text...
```

produces as output:

```
one  This is the first paragraph.  Notice how the first line of the resulting paragraph lines
     up with the other lines in the paragraph.

two  And here we are at the second paragraph already.  You may notice that the argu-
     ment to .ip appears in the margin.
We can continue text without starting a new indented paragraph by using the .lp request.
```

If you have spaces in the label of a **.ip** request, you must use an "unpaddable space" instead of a regular space. This is typed as a backslash character ("\") followed by a space. For example, to print the label "Part 1", enter:

```
.ip "Part\ 1"
```

If a label of an indented paragraph (that is, the argument to **.ip**) is longer than the space allocated for the label, **.ip** will begin a new line after the label. For example, the input:

```
.ip longlabel
This paragraph had a long label.
The first character of text on the first line
will not line up with the text on second and subsequent lines,
although they will line up with each other.
```

will produce:

```
longlabel
     This paragraph had a long label.  The first character of text on the first line will not
     line up with the text on second and subsequent lines, although they will line up
     with each other.
```

It is possible to change the size of the label by using a second argument which is the size of the label. For example, the above example could be done correctly by saying:

```
.ip longlabel 10
```

which will make the paragraph indent 10 spaces for this paragraph only. If you have many paragraphs to indent all the same amount, use the *number register* **ii**. For example, to leave one inch of space for the label, type:

```
.nr ii 1i
```

somewhere before the first call to **.ip**. Refer to the reference manual for more information.

If **.ip** is used with no argument at all no hanging tag will be printed. For example, the input:

```
.ip [a]
This is the first paragraph of the example.
We have seen this sort of example before.
.ip
This paragraph is lined up with the previous paragraph,
but it has no tag in the margin.
```

produces as output:

[a] This is the first paragraph of the example. We have seen this sort of example before.

This paragraph is lined up with the previous paragraph, but it has no tag in the margin.

A special case of **.ip** is **.np**, which automatically numbers paragraphs sequentially from 1. The numbering is reset at the next **.pp**, **.lp**, or **.sh** (to be described in the next section) request. For example, the input:

```
.np
This is the first point.
.np
This is the second point.
Points are just regular paragraphs
which are given sequence numbers automatically
by the .np request.
.pp
This paragraph will reset numbering by .np.
.np
For example,
we have reverted to numbering from one now.
```

generates:

(1) This is the first point.

(2) This is the second point. Points are just regular paragraphs which are given sequence numbers automatically by the .np request.

This paragraph will reset numbering by .np.

(1) For example, we have reverted to numbering from one now.

5.2. Section Headings

Section numbers (such as the ones used in this document) can be automatically generated using the **.sh** request. You must tell **.sh** the *depth* of the section number and a section title. The depth specifies how many numbers are to appear (separated by decimal points) in the section number. For example, the section number **4.2.5** has a depth of three.

Section numbers are incremented in a fairly intuitive fashion. If you add a number (increase the depth), the new number starts out at one. If you subtract section numbers (or keep the same number) the final number is incremented. For example, the input:

```
.sh 1 "The Preprocessor"
.sh 2 "Basic Concepts"
.sh 2 "Control Inputs"
.sh 3
.sh 3
.sh 1 "Code Generation"
.sh 3
```

produces as output the result:

```
1. The Preprocessor
   1.1. Basic Concepts
   1.2. Control Inputs
      1.2.1.
      1.2.2.
2. Code Generation
      2.1.1.
```

You can specify the section number to begin by placing the section number after the section title, using spaces instead of dots. For example, the request:

```
.sh 3 "Another section" 7 3 4
```

will begin the section numbered **7.3.4**; all subsequent **.sh** requests will number relative to this number.

There are more complex features which will cause each section to be indented proportionally to the depth of the section. For example, if you enter:

.nr si *N*

each section will be indented by an amount *N*. *N* must have a scaling factor attached, that is, it must be of the form *Nx*, where *x* is a character telling what units *N* is in. Common values for *x* are **i** for inches, **c** for centimeters, and **n** for *ens* (the width of a single character). For example, to indent each section one-half inch, type:

```
.nr si 0.5i
```

After this, sections will be indented by one-half inch per level of depth in the section number. For example, this document was produced using the request

```
.nr si 3n
```

at the beginning of the input file, giving three spaces of indent per section depth.

Section headers without automatically generated numbers can be done using:

```
.uh "Title"
```

which will do a section heading, but will put no number on the section.

5.3. Parts of the Basic Paper

There are some requests which assist in setting up papers. The **.tp** request initializes for a title page. There are no headers or footers on a title page, and unlike other pages you can space down and leave blank space at the top. For example, a typical title page might appear as:

```
.tp
.sp 2i
.(l C
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank N. Furter
.)l
.bp
```

The request **.th** sets up the environment of the NROFF processor to do a thesis, using the rules established at Berkeley. It defines the correct headers and footers (a page number in the upper right hand corner only), sets the margins correctly, and double spaces.

The **.+c** *T* request can be used to start chapters. Each chapter is automatically numbered from one, and a heading is printed at the top of each chapter with the chapter number and the chapter name *T*. For example, to begin a chapter called "Conclusions", use the request:

```
.+c "CONCLUSIONS"
```

which will produce, on a new page, the lines

CHAPTER 5
CONCLUSIONS

with appropriate spacing for a thesis. Also, the header is moved to the foot of the page on the first page of a chapter. Although the **.+c** request was not designed to work only with the **.th** request, it is tuned for the format acceptable for a PhD thesis at Berkeley.

If the title parameter *T* is omitted from the **.+c** request, the result is a chapter with no heading. This can also be used at the beginning of a paper; for example, **.+c** was used to generate page one of this document.

Although papers traditionally have the abstract, table of contents, and so forth at the front of the paper, it is more convenient to format and print them last when using NROFF. This is so that index entries can be collected and then printed for the table of contents (or whatever). At the end of the paper, issue the **.++ P** request, which begins the preliminary part of the paper. After issuing this request, the **.+c** request will begin a preliminary section of the paper. Most notably, this prints the page number restarted from one in lower case Roman numbers. **.+c** may be used repeatedly to begin different parts of the front material for example, the abstract, the table of contents, acknowledgments, list of illustrations, etc. The request **.++ B** may also be used to begin the bibliographic section at the end of the paper. For example, the paper might appear as outlined in figure 2. (In this figure, comments begin with the sequence \".)

5.4. Equations and Tables

Two special UNIX programs exist to format special types of material. **Eqn** and **neqn** set equations for the phototypesetter and NROFF respectively. **Tbl** arranges to print extremely pretty tables in a variety of formats. This document will only describe the embellishments to the standard features; consult the reference manuals for those processors for a description of their use.

The **eqn** and **neqn** programs are described fully in the document *Typesetting Mathematics — Users' Guide* by Brian W. Kernighan and Lorinda L. Cherry. Equations are centered, and are kept on one page. They are introduced by the **.EQ** request and terminated by the **.EN** request.

The **.EQ** request may take an equation number as an optional argument, which is printed vertically centered on the right hand side of the equation. If the equation becomes too long it should be split between two lines. To do this, type:

```
.EQ (eq 34)
text of equation 34
.EN C
.EQ
continuation of equation 34
.EN
```

The **C** on the **.EN** request specifies that the equation will be continued.

The **tbl** program produces tables. It is fully described (including numerous examples) in the document *Tbl — A Program to Format Tables* by M. E. Lesk. Tables begin with the **.TS** request and end with the **.TE** request. Tables are normally kept on a single page. If you have a table which is too big to fit on a single page, so that you know it will extend to several pages, begin the table with the request **.TS H** and put the request **.TH**

```
.th                            \" set for thesis mode
.fo "DRAFT"                    \" define footer for each page
.tp                            \" begin title page
.(l C                          \" center a large block
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank Furter
.)l                            \" end centered part
.+c INTRODUCTION               \" begin chapter named "INTRODUCTION"
.(x t                          \" make an entry into index 't'
Introduction
.)x                            \" end of index entry
text of chapter one
.+c "NEXT CHAPTER"             \" begin another chapter
.(x t                          \" enter into index 't' again
Next Chapter
.)x
text of chapter two
.+c CONCLUSIONS
.(x t
Conclusions
.)x
text of chapter three
.++ B                          \" begin bibliographic information
.+c BIBLIOGRAPHY               \" begin another 'chapter'
.(x t
Bibliography
.)x
text of bibliography
.++ P                          \" begin preliminary material
.+c "TABLE OF CONTENTS"
.xp t                          \" print index 't' collected above
.+c PREFACE                    \" begin another preliminary section
text of preface
```

Figure 2. Outline of a Sample Paper

after the part of the table which you want duplicated at the top of every page that the table is printed on. For example, a table definition for a long table might look like:

```
.TS H
c s s
n n n.
THE TABLE TITLE
.TH
text of the table
.TE
```

5.5. Two Column Output

You can get two column output automatically by using the request **.2c**. This causes everything after it to be output in two-column form. The request **.bc** will start a new column; it differs from **.bp** in that **.bp** may leave a totally blank column when it starts a new page. To revert to single column output, use **.1c**.

5.6. Defining Macros

A *macro* is a collection of requests and text which may be used by stating a simple request. Macros begin with the line **.de** *xx* (where *xx* is the name of the macro to be defined) and end with the line consisting of two dots. After defining the macro, stating the line .*xx* is the same as stating all the other lines. For example, to define a macro that spaces 3 lines and then centers the next input line, enter:

```
.de SS
.sp 3
.ce
..
```

and use it by typing:

```
.SS
Title Line
(beginning of text)
```

Macro names may be one or two characters. In order to avoid conflicts with names in −me, always use upper case letters as names. The only names to avoid are **TS**, **TH**, **TE**, **EQ**, and **EN**.

5.7. Annotations Inside Keeps

Sometimes you may want to put a footnote or index entry inside a keep. For example, if you want to maintain a "list of figures" you will want to do something like:

```
.(z
.(c
text of figure
.)c
.ce
Figure 5.
.(x f
Figure 5
.)x
.)z
```

which you may hope will give you a figure with a label and an entry in the index **f** (presumably a list of figures index). Unfortunately, the index entry is read and interpreted when the keep is read, not when it is printed, so the page number in the index is likely to be wrong. The solution is to use the magic string \! at the beginning of all the lines dealing with the index. In other words, you should use:

```
.(z
.(c
Text of figure
.)c
.ce
Figure 5.
\!.(x f
\!Figure 5
\!.)x
.)z
```

which will defer the processing of the index until the figure is output. This will guarantee that the page number in the index is correct. The same comments apply to blocks (with **.(b** and **.)b**) as well.

6. TROFF and the Photosetter

With a little care, you can prepare documents that will print nicely on either a regular terminal or when phototypeset using the TROFF formatting program.

6.1. Fonts

A *font* is a style of type. There are three fonts that are available simultaneously, Times Roman, Times Italic, and Times Bold, plus the special math font. The normal font is Roman. Text which would be underlined in NROFF with the **.ul** request is set in italics in TROFF.

There are ways of switching between fonts. The requests **.r**, **.i**, and **.b** switch to Roman, italic, and bold fonts respectively. You can set a single word in some font by typing (for example):

```
.i word
```

which will set *word* in italics but does not affect the surrounding text. In NROFF, italic and bold text is underlined.

Notice that if you are setting more than one word in whatever font, you must surround that word with double quote marks (‘"’) so that it will appear to the NROFF processor as a single word. The quote marks will not appear in the formatted text. If you do want a quote mark to appear, you should quote the entire string (even if a single word), and use *two* quote marks where you want one to appear. For example, if you want to produce the text:

"*Master Control*"

in italics, you must type:

```
.i """Master Control\|"""
```

The \| produces a very narrow space so that the "l" does not overlap the quote sign in TROFF, like this:

"*Master Control*"

There are also several "pseudo-fonts" available. The input:

```
.(b
.u underlined
.bi "bold italics"
.bx "words in a box"
.)b
```

generates

underlined
bold italics
$\boxed{\text{words in a box}}$

In NROFF these all just underline the text. Notice that pseudo font requests set only the single parameter in the pseudo font; ordinary font requests will begin setting all text in the special font if you do not provide a parameter. No more than one word should appear with these three font requests in the middle of lines. This is because of the way TROFF justifies text. For example, if you were to issue the requests:

```
.bi "some bold italics"
and
.bx "words in a box"
```

in the middle of a line TROFF would produce ***ssoommee bboolldd iittaalliiccss*** and |$\boxed{\text{words in a box}}$, which I think you will agree does not look good.

The second parameter of all font requests is set in the original font. For example, the font request:

```
.b bold face
```

generates "bold" in bold font, but sets "face" in the font of the surrounding text, resulting in:

boldface.

To set the two words **bold** and **face** both in **bold face**, type:

```
.b "bold face"
```

You can mix fonts in a word by using the special sequence **\c** at the end of a line to indicate "continue text processing"; this allows input lines to be joined together without a space inbetween them. For example, the input:

```
.u under \c
.i italics
```

generates under*italics*, but if we had typed:

```
.u under
.i italics
```

the result would have been under *italics* as two words.

6.2. Point Sizes

The phototypesetter supports different sizes of type, measured in points. The default point size is 10 points for most text, 8 points for footnotes. To change the pointsize, type:

.sz $+N$

where N is the size wanted in points. The *vertical spacing* (distance between the bottom of most letters (the *baseline*) between adjacent lines) is set to be proportional to the type size.

Warning: changing point sizes on the phototypesetter is a slow mechanical operation. Size changes should be considered carefully.

6.3. Quotes

It is conventional when using the typesetter to use pairs of grave and acute accents to generate double quotes, rather than the double quote character (‘"’). This is because it looks better to use grave and acute accents; for example, compare "quote" to “quote”.

In order to make quotes compatible between the typesetter and terminals, you may use the sequences **\*(lq** and **\*(rq** to stand for the left and right quote respectively.

These both appear as " on most terminals, but are typeset as “ and ” respectively. For example, use:

```
\*(lqSome things aren't true
even if they did happen.\*(rq
```

to generate the result:

“Some things aren't true even if they did happen.”

As a shorthand, the special font request:

```
.q "quoted text"
```

will generate “quoted text”. Notice that you must surround the material to be quoted with double quote marks if it is more than one word.

Acknowledgments

I would like to thank Bob Epstein, Bill Joy, and Larry Rowe for having the courage to use the −me macros to produce non-trivial papers during the development stages; Ricki Blau, Pamela Humphrey, and Jim Joyce for their help with the documentation phase; and the plethora of people who have contributed ideas and have given support for the project.

This document was TROFF'ed on December 18, 1979 and applies to version 1.1 of the −me macros.

−ME REFERENCE MANUAL

Release 1.1/25

Eric P. Allman

Electronics Research Laboratory
University of California, Berkeley
Berkeley, California 94720

This document describes in extremely terse form the features of the **−me** macro package for version seven NROFF/TROFF. Some familiarity is assumed with those programs, specifically, the reader should understand breaks, fonts, pointsizes, the use and definition of number registers and strings, how to define macros, and scaling factors for ens, points, v's (vertical line spaces), etc.

For a more casual introduction to text processing using NROFF, refer to the document *Writing Papers with NROFF using −me.*

There are a number of macro parameters that may be adjusted. Fonts may be set to a font number only. In NROFF font 8 is underlined, and is set in bold font in TROFF (although font 3, bold in TROFF, is not underlined in NROFF). Font 0 is no font change; the font of the surrounding text is used instead. Notice that fonts 0 and 8 are "pseudo-fonts"; that is, they are simulated by the macros. This means that although it is legal to set a font register to zero or eight, it is not legal to use the escape character form, such as:

```
\f8
```

All distances are in basic units, so it is nearly always necessary to use a scaling factor. For example, the request to set the paragraph indent to eight one-en spaces is:

```
.nr pi 8n
```

and not

```
.nr pi 8
```

which would set the paragraph indent to eight basic units, or about 0.02 inch. Default parameter values are given in brackets in the remainder of this document.

Registers and strings of the form **$***x* may be used in expressions but should not be changed. Macros of the form **$***x* perform some function (as described) and may be redefined to change this function. This may be a sensitive operation; look at the body of the original macro before changing it.

All names in −me follow a rigid naming convention. The user may define number registers, strings, and macros, provided that s/he uses single character upper case names or double character names consisting of letters and digits, with at least one upper case letter. In no case should special characters be used in user-defined names.

On daisy wheel type printers in twelve pitch, the **−rx1** flag can be stated to make lines default to one eighth inch (the normal spacing for a newline in twelve-pitch). This is normally

†NROFF and TROFF are Trademarks of Bell Laboratories.

too small for easy readability, so the default is to space one sixth inch.

This documentation was TROFF'ed on December 14, 1979 and applies to version 1.1/25 of the —me macros.

1. Paragraphing

These macros are used to begin paragraphs. The standard paragraph macro is **.pp**; the others are all variants to be used for special purposes.

The first call to one of the paragraphing macros defined in this section or the **.sh** macro (defined in the next session) *initializes* the macro processor. After initialization it is not possible to use any of the following requests: **.sc**, **.lo**, **.th**, or **.ac**. Also, the effects of changing parameters which will have a global effect on the format of the page (notably page length and header and footer margins) are not well defined and should be avoided.

.lp	Begin left-justified paragraph. Centering and underlining are turned off if they were on, the font is set to **\n(pf** [1] the type size is set to **\n(pp** [10p], and a **\n(ps** space is inserted before the paragraph [0.35v in TROFF, 1v or 0.5v in NROFF depending on device resolution]. The indent is reset to **\n($i** [0] plus **\n(po** [0] unless the paragraph is inside a display. (see **.ba**). At least the first two lines of the paragraph are kept together on a page.
.pp	Like **.lp**, except that it puts **\n(pi** [5n] units of indent. This is the standard paragraph macro.
.ip *T I*	Indented paragraph with hanging tag. The body of the following paragraph is indented *I* spaces (or **\n(ii** [5n] spaces if *I* is not specified) more than a non-indented paragraph (such as with **.pp**) is. The title *T* is exdented (opposite of indented). The result is a paragraph with an even left edge and *T* printed in the margin. Any spaces in *T* must be unpaddable. If *T* will not fit in the space provided, **.ip** will start a new line.
.np	A variant of .ip which numbers paragraphs. Numbering is reset after a **.lp**, **.pp**, or **.sh**. The current paragraph number is in **\n($p**.

2. Section Headings

Numbered sections are similiar to paragraphs except that a section number is automatically generated for each one. The section numbers are of the form **1.2.3**. The *depth* of the section is the count of numbers (separated by decimal points) in the section number.

Unnumbered section headings are similar, except that no number is attached to the heading.

.sh *+N T a b c d e f*	Begin numbered section of depth *N*. If *N* is missing the current depth (maintained in the number register **\n($0**) is used. The values of the individual parts of the section number are maintained in **\n($1** through **\n($6**. There is a **\n(ss** [1v] space before the section. *T* is printed as a section title in font **\n(sf** [8] and size **\n(sp** [10p]. The "name" of the section may be accessed via ***($n**. If **\n(si** is non-zero, the base indent is set to **\n(si** times the section depth, and the section title is exdented. (See **.ba**.) Also, an additional indent of **\n(so** [0] is added to the section title (but not to the body of the section). The font is then set to the paragraph font, so that more information may occur on the line with the section number and title. **.sh** insures that there is enough room to print the section head plus the beginning of a paragraph (about 3 lines total). If *a* through *f* are specified, the section number is set to that number rather than incremented automatically. If any of *a* through *f* are a hyphen that number is not reset. If *T* is a single

	underscore ("_") then the section depth and numbering is reset, but the base indent is not reset and nothing is printed out. This is useful to automatically coordinate section numbers with chapter numbers.
.sx +*N*	Go to section depth *N* [**−1**], but do not print the number and title, and do not increment the section number at level *N*. This has the effect of starting a new paragraph at level *N*.
.uh *T*	Unnumbered section heading. The title *T* is printed with the same rules for spacing, font, etc., as for **.sh**.
.$p *T B N*	Print section heading. May be redefined to get fancier headings. *T* is the title passed on the **.sh** or **.uh** line; *B* is the section number for this section, and *N* is the depth of this section. These parameters are not always present; in particular, **.sh** passes all three, **.uh** passes only the first, and **.sx** passes three, but the first two are null strings. Care should be taken if this macro is redefined; it is quite complex and subtle.
.$0 *T B N*	This macro is called automatically after every call to **.$p**. It is normally undefined, but may be used to automatically put every section title into the table of contents or for some similiar function. *T* is the section title for the section title which was just printed, *B* is the section number, and *N* is the section depth.
.$1 − .$6	Traps called just before printing that depth section. May be defined to (for example) give variable spacing before sections. These macros are called from **.$p**, so if you redefine that macro you may lose this feature.

3. Headers and Footers

Headers and footers are put at the top and bottom of every page automatically. They are set in font **\n(tf** [3] and size **\n(tp** [10p]. Each of the definitions apply as of the *next* page. Three-part titles must be quoted if there are two blanks adjacent anywhere in the title or more than eight blanks total.

The spacing of headers and footers are controlled by three number registers. **\n(hm** [4v] is the distance from the top of the page to the top of the header, **\n(fm** [3v] is the distance from the bottom of the page to the bottom of the footer, **\n(tm** [7v] is the distance from the top of the page to the top of the text, and **\n(bm** [6v] is the distance from the bottom of the page to the bottom of the text (nominal). The macros **.m1**, **.m2**, **.m3**, and **.m4** are also supplied for compatibility with ROFF documents.

.he *'l'm'r'*	Define three-part header, to be printed on the top of every page.
.fo *'l'm'r'*	Define footer, to be printed at the bottom of every page.
.eh *'l'm'r'*	Define header, to be printed at the top of every even-numbered page.
.oh *'l'm'r'*	Define header, to be printed at the top of every odd-numbered page.
.ef *'l'm'r'*	Define footer, to be printed at the bottom of every even-numbered page.
.of *'l'm'r'*	Define footer, to be printed at the bottom of every odd-numbered page.
.hx	Suppress headers and footers on the next page.
.m1 +*N*	Set the space between the top of the page and the header [4v].
.m2 +*N*	Set the space between the header and the first line of text [2v].
.m3 +*N*	Set the space between the bottom of the text and the footer [2v].
.m4 +*N*	Set the space between the footer and the bottom of the page [4v].
.ep	End this page, but do not begin the next page. Useful for forcing out footnotes, but other than that hardly every used. Must be followed by

	a **.bp** or the end of input.
.$h	Called at every page to print the header. May be redefined to provide fancy (e.g., multi-line) headers, but doing so loses the function of the **.he**, **.fo**, **.eh**, **.oh**, **.ef**, and **.of** requests, as well as the chapter-style title feature of **.+c**.
.$f	Print footer; same comments apply as in **.$h**.
.$H	A normally undefined macro which is called at the top of each page (after outputting the header, initial saved floating keeps, etc.); in other words, this macro is called immediately before printing text on a page. It can be used for column headings and the like.

4. Displays

All displays except centered blocks and block quotes are preceeded and followed by an extra **\n(bs** [same as **\n(ps**] space. Quote spacing is stored in a separate register; centered blocks have no default initial or trailing space. The vertical spacing of all displays except quotes and centered blocks is stored in register **\n($R** instead of **\n($r**.

.(l *m f*	Begin list. Lists are single spaced, unfilled text. If *f* is **F**, the list will be filled. If *m* [**I**] is **I** the list is indented by **\n(bi** [4n]; if **M** the list is indented to the left margin; if **L** the list is left justified with respect to the text (different from **M** only if the base indent (stored in **\n($i** and set with **.ba**) is not zero); and if **C** the list is centered on a line-by-line basis. The list is set in font **\n(df** [0]. Must be matched by a **.)l**. This macro is almost like **.(b** except that no attempt is made to keep the display on one page.
.)l	End list.
.(q	Begin major quote. These are single spaced, filled, moved in from the text on both sides by **\n(qi** [4n], preceeded and followed by **\n(qs** [same as **\n(bs**] space, and are set in point size **\n(qp** [one point smaller than surrounding text].
.)q	End major quote.
.(b *m f*	Begin block. Blocks are a form of *keep*, where the text of a keep is kept together on one page if possible (keeps are useful for tables and figures which should not be broken over a page). If the block will not fit on the current page a new page is begun, *unless* that would leave more than **\n(bt** [0] white space at the bottom of the text. If **\n(bt** is zero, the threshold feature is turned off. Blocks are not filled unless *f* is **F**, when they are filled. The block will be left-justified if *m* is **L**, indented by **\n(bi** [4n] if *m* is **I** or absent, centered (line-for-line) if *m* is **C**, and left justified to the margin (not to the base indent) if *m* is **M**. The block is set in font **\n(df** [0].
.)b	End block.
.(z *m f*	Begin floating keep. Like **.(b** except that the keep is *floated* to the bottom of the page or the top of the next page. Therefore, its position relative to the text changes. The floating keep is preceeded and followed by **\n(zs** [1v] space. Also, it defaults to mode **M**.
.)z	End floating keep.
.(c	Begin centered block. The next keep is centered as a block, rather than on a line-by-line basis as with **.(b C**. This call may be nested inside keeps.

.)c	End centered block.

5. Annotations

.(d	Begin delayed text. Everything in the next keep is saved for output later with **.pd**, in a manner similar to footnotes.
.)d *n*	End delayed text. The delayed text number register **\n($d** and the associated string **\*#** are incremented if **\*#** has been referenced.
.pd	Print delayed text. Everything diverted via **.(d** is printed and truncated. This might be used at the end of each chapter.
.(f	Begin footnote. The text of the footnote is floated to the bottom of the page and set in font **\n(ff** [1] and size **\n(fp** [8p]. Each entry is preceeded by **\n(fs** [0.2v] space, is indented **\n(fi** [3n] on the first line, and is indented **\n(fu** [0] from the right margin. Footnotes line up underneath two columned output. If the text of the footnote will not all fit on one page it will be carried over to the next page.
.)f *n*	End footnote. The number register **\n($f** and the associated string **\**** are incremented if they have been referenced.
.$s	The macro to output the footnote seperator. This macro may be redefined to give other size lines or other types of separators. Currently it draws a 1.5i line.
.(x *x*	Begin index entry. Index entries are saved in the index *x* [x] until called up with **.xp.** Each entry is preceeded by a **\n(xs** [0.2v] space. Each entry is "undented" by **\n(xu** [0.5i]; this register tells how far the page number extends into the right margin.
.)x *P A*	End index entry. The index entry is finished with a row of dots with *A* [null] right justified on the last line (such as for an author's name), followed by P [**\n%**]. If *A* is specified, *P* must be specified; **\n%** can be used to print the current page number. If *P* is an underscore, no page number and no row of dots are printed.
.xp *x*	Print index *x* [x]. The index is formated in the font, size, and so forth in effect at the time it is printed, rather than at the time it is collected.

6. Columned Output

.2c *+S N*	Enter two-column mode. The column separation is set to *+S* [4n, 0.5i in ACM mode] (saved in **\n($s**). The column width, calculated to fill the single column line length with both columns, is stored in **\n($l**. The current column is in **\n($c**. You can test register **\n($m** [1] to see if you are in single column or double column mode. Actually, the request enters *N* [2] columned output.
.1c	Revert to single-column mode.
.bc	Begin column. This is like **.bp** except that it begins a new column on a new page only if necessary, rather than forcing a whole new page if there is another column left on the current page.

7. Fonts and Sizes

.sz *+P*	The pointsize is set to *P* [10p], and the line spacing is set proportionally. The ratio of line spacing to pointsize is stored in **\n($r**. The ratio used internally by displays and annotations is stored in **\n($R** (although this is not used by **.sz**).

.r *W X*	Set *W* in roman font, appending *X* in the previous font. To append different font requests, use *X* = **\c**. If no parameters, change to roman font.
.i *W X*	Set *W* in italics, appending *X* in the previous font. If no parameters, change to italic font. Underlines in NROFF.
.b *W X*	Set *W* in bold font and append *X* in the previous font. If no parameters, switch to bold font. In NROFF, underlines.
.rb *W X*	Set *W* in bold font and append *X* in the previous font. If no parameters, switch to bold font. **.rb** differs from **.b** in that **.rb** does not underline in NROFF.
.u *W X*	Underline *W* and append *X*. This is a true underlining, as opposed to the **.ul** request, which changes to "underline font" (usually italics in TROFF). It won't work right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.
.q *W X*	Quote *W* and append *X*. In NROFF this just surrounds *W* with double quote marks (‘"’), but in TROFF uses directed quotes.
.bi *W X*	Set *W* in bold italics and append *X*. Actually, sets *W* in italic and overstrikes once. Underlines in NROFF. It won't work right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.
.bx *W X*	Sets *W* in a box, with *X* appended. Underlines in NROFF. It won't work right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

8. Roff Support

.ix +*N*	Indent, no break. Equivalent to **'in** *N*.
.bl *N*	Leave *N* contiguous white space, on the next page if not enough room on this page. Equivalent to a **.sp** *N* inside a block.
.pa +*N*	Equivalent to **.bp**.
.ro	Set page number in roman numerals. Equivalent to **.af % i**.
.ar	Set page number in arabic. Equivalent to **.af % 1**.
.n1	Number lines in margin from one on each page.
.n2 *N*	Number lines from *N*, stop if *N* = 0.
.sk	Leave the next output page blank, except for headers and footers. This is used to leave space for a full-page diagram which is produced externally and pasted in later. To get a partial-page paste-in display, say .sv *N*, where *N* is the amount of space to leave; this space will be output immediately if there is room, and will otherwise be output at the top of the next page. However, be warned: if *N* is greater than the amount of available space on an empty page, no space will ever be output.

9. Preprocessor Support

.EQ *m T*	Begin equation. The equation is centered if *m* is **C** or omitted, indented **\n(bi** [4n] if *m* is **I**, and left justified if *m* is **L**. *T* is a title printed on the right margin next to the equation. See *Typesetting Mathematics — User's Guide* by Brian W. Kernighan and Lorinda L. Cherry.

.EN *c*	End equation. If *c* is **C** the equation must be continued by immediately following with another **.EQ**, the text of which can be centered along with this one. Otherwise, the equation is printed, always on one page, with **\n(es** [0.5v in TROFF, 1v in NROFF] space above and below it.
.TS *h*	Table start. Tables are single spaced and kept on one page if possible. If you have a large table which will not fit on one page, use *h* = **H** and follow the header part (to be printed on every page of the table) with a **.TH**. See *Tbl − A Program to Format Tables* by M. E. Lesk.
.TH	With **.TS H**, ends the header portion of the table.
.TE	Table end. Note that this table does not float, in fact, it is not even guaranteed to stay on one page if you use requests such as **.sp** intermixed with the text of the table. If you want it to float (or if you use requests inside the table), surround the entire table (including the **.TS** and **.TE** requests) with the requests **.(z** and **.)z**.

10. Miscellaneous

.re	Reset tabs. Set to every 0.5i in TROFF and every 0.8i in NROFF.
.ba +*N*	Set the base indent to +*N* [0] (saved in **\n($i**). All paragraphs, sections, and displays come out indented by this amount. Titles and footnotes are unaffected. The **.sh** request performs a **.ba** request if **\n(si** [0] is not zero, and sets the base indent to **\n(si*\n($0**.
.xl +*N*	Set the line length to *N* [6.0i]. This differs from **.ll** because it only affects the current environment.
.ll +*N*	Set line length in all environments to *N* [6.0i]. This should not be used after output has begun, and particularly not in two-columned output. The current line length is stored in **\n($l**.
.hl	Draws a horizontal line the length of the page. This is useful inside floating keeps to differentiate between the text and the figure.
.lo	This macro loads another set of macros (in **/usr/lib/me/local.me**) which is intended to be a set of locally defined macros. These macros should all be of the form **.****X*, where *X* is any letter (upper or lower case) or digit.

11. Standard Papers

.tp	Begin title page. Spacing at the top of the page can occur, and headers and footers are supressed. Also, the page number is not incremented for this page.
.th	Set thesis mode. This defines the modes acceptable for a doctoral dissertation at Berkeley. It double spaces, defines the header to be a single page number, and changes the margins to be 1.5 inch on the left and one inch on the top. **.++** and **.+c** should be used with it. This macro must be stated before initialization, that is, before the first call of a paragraphing macro or **.sh**.
.++ *m H*	This request defines the section of the paper which we are entering. The section type is defined by *m*. **C** means that we are entering the chapter portion of the paper, **A** means that we are entering the appendix portion of the paper, **P** means that the material following should be the preliminary portion (abstract, table of contents, etc.) portion of the paper, **AB** means that we are entering the abstract (numbered independently from 1 in Arabic numerals), and **B** means that we are entering the bibliographic portion at the end of the paper. Also, the variants **RC**

and **RA** are allowed, which specify renumbering of pages from one at the beginning of each chapter or appendix, respectively. The *H* parameter defines the new header. If there are any spaces in it, the entire header must be quoted. If you want the header to have the chapter number in it, Use the string **n(ch**. For example, to number appendixes **A.1** etc., type **.++ RA** '''\\n(ch.%'. Each section (chapter, appendix, etc.) should be preceeded by the **.+c** request. It should be mentioned that it is easier when using TROFF to put the front material at the end of the paper, so that the table of contents can be collected and output; this material can then be physically moved to the beginning of the paper.

.+c *T*	Begin chapter with title *T*. The chapter number is maintained in **\n(ch**. This register is incremented every time **.+c** is called with a parameter. The title and chapter number are printed by **.$c**. The header is moved to the footer on the first page of each chapter. If *T* is omitted, **.$c** is not called; this is useful for doing your own "title page" at the beginning of papers without a title page proper. **.$c** calls **.$C** as a hook so that chapter titles can be inserted into a table of contents automatically. The footnote numbering is reset to one.
.$c *T*	Print chapter number (from **\n(ch**) and *T*. This macro can be redefined to your liking. It is defined by default to be acceptable for a PhD thesis at Berkeley. This macro calls **$C**, which can be defined to make index entries, or whatever.
.$C *K N T*	This macro is called by **.$c**. It is normally undefined, but can be used to automatically insert index entries, or whatever. *K* is a keyword, either "Chapter" or "Appendix" (depending on the **.++** mode); *N* is the chapter or appendix number, and *T* is the chapter or appendix title.
.ac *A N*	This macro (short for **.acm**) sets up the NROFF environment for photo-ready papers as used by the ACM. This format is 25% larger, and has no headers or footers. The author's name *A* is printed at the bottom of the page (but off the part which will be printed in the conference proceedings), together with the current page number and the total number of pages *N*. Additionally, this macro loads the file **/usr/lib/me/acm.me**, which may later be augmented with other macros useful for printing papers for ACM conferences. It should be noted that this macro will not work correctly in TROFF, since it sets the page length wider than the physical width of the phototypesetter roll.

12. Predefined Strings

**	Footnote number, actually ***[\n($f*]**. This macro is incremented after each call to **.)f**.
*#	Delayed text number. Actually **[\n($d]**.
*[	Superscript. This string gives upward movement and a change to a smaller point size if possible, otherwise it gives the left bracket character ('**[**'). Extra space is left above the line to allow room for the superscript.
*]	Unsuperscript. Inverse to ***[**. For example, to produce a superscript you might type **x*[2*]**, which will produce $\mathbf{x}^{2}$.
*<	Subscript. Defaults to '<' if half-carriage motion not possible. Extra space is left below the line to allow for the subscript.
*>	Inverse to ***<**.

***(dw**	The day of the week, as a word.
***(mo**	The month, as a word.
***(td**	Today's date, directly printable. The date is of the form December 14, 1979. Other forms of the date can be used by using **\n(dy** (the day of the month; for example, 14), ***(mo** (as noted above) or **\n(mo** (the same, but as an ordinal number; for example, December is 12), and **\n(yr** (the last two digits of the current year).
***(lq**	Left quote marks. Double quote in NROFF.
***(rq**	Right quote.
***−**	¾ em dash in TROFF; two hyphens in NROFF.

13. Special Characters and Marks

There are a number of special characters and diacritical marks (such as accents) available through −me. To reference these characters, you must call the macro **.sc** to define the characters before using them.

.sc	Define special characters and diacritical marks, as described in the remainder of this section. This macro must be stated before initialization.

The special characters available are listed below.

Name	Usage	Example	
Acute accent	*'	a*'	á
Grave accent	*`	e*`	è
Umlat	*:	u*:	ü
Tilde	*~	n*~	ñ
Caret	*^	e*^	ê
Cedilla	*,	c*,	ç
Czech	*v	e*v	ě
Circle	*o	A*o	Å
There exists	*(qe		∃
For all	*(qa		∀

Acknowledgments

I would like to thank Bob Epstein, Bill Joy, and Larry Rowe for having the courage to use the −me macros to produce non-trivial papers during the development stages; Ricki Blau, Pamela Humphrey, and Jim Joyce for their help with the documentation phase; and the plethora of people who have contributed ideas and have given support for the project.

A System for Typesetting Mathematics

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the design and implementation of a system for typesetting mathematics. The language has been designed to be easy to learn and to use by people (for example, secretaries and mathematical typists) who know neither mathematics nor typesetting. Experience indicates that the language can be learned in an hour or so, for it has few rules and fewer exceptions. For typical expressions, the size and font changes, positioning, line drawing, and the like necessary to print according to mathematical conventions are all done automatically. For example, the input

sum from i=0 to infinity x sub i = pi over 2

produces

$$\sum_{i=0}^{\infty} x_i = \frac{\pi}{2}$$

The syntax of the language is specified by a small context-free grammar; a compiler-compiler is used to make a compiler that translates this language into typesetting commands. Output may be produced on either a phototypesetter or on a terminal with forward and reverse half-line motions. The system interfaces directly with text formatting programs, so mixtures of text and mathematics may be handled simply.

This paper is a revision of a paper originally published in CACM, March, 1975.

1. Introduction

"Mathematics is known in the trade as *difficult*, or *penalty, copy* because it is slower, more difficult, and more expensive to set in type than any other kind of copy normally occurring in books and journals." [1]

One difficulty with mathematical text is the multiplicity of characters, sizes, and fonts. An expression such as

$$\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$$

requires an intimate mixture of roman, italic and greek letters, in three sizes, and a special character or two. ("Requires" is perhaps the wrong word, but mathematics has its own typographical conventions which are quite different from those of ordinary text.) Typesetting such an expression by traditional methods is still an essentially manual operation.

A second difficulty is the two dimensional character of mathematics, which the superscript and limits in the preceding example showed in its simplest form. This is carried further by

$$a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cdots}}}$$

and still further by

$$\int \frac{dx}{ae^{mx} - be^{-mx}} = \left\{ \begin{array}{l} \dfrac{1}{2m\sqrt{ab}} \log \dfrac{\sqrt{a}\,e^{mx} - \sqrt{b}}{\sqrt{a}\,e^{mx} + \sqrt{b}} \\ \dfrac{1}{m\sqrt{ab}} \tanh^{-1}(\dfrac{\sqrt{a}}{\sqrt{b}} e^{mx}) \\ \dfrac{-1}{m\sqrt{ab}} \coth^{-1}(\dfrac{\sqrt{a}}{\sqrt{b}} e^{mx}) \end{array} \right.$$

These examples also show line-drawing, built-up characters like braces and radicals, and a spectrum of positioning problems. (Section 6 shows what a user has to type to produce these on our system.)

2. Photocomposition

Photocomposition techniques can be used to solve some of the problems of typesetting mathematics. A phototypesetter is a device which exposes a piece of photographic paper or film, placing characters wherever they are wanted. The Graphic Systems phototypesetter[2] on the UNIX operating system[3] works by shining light through a character stencil. The character is made the right size by lenses, and the light beam directed by fiber optics to the desired place on a piece of photographic paper. The exposed paper is developed and typically used in some form of photo-offset reproduction.

On UNIX, the phototypesetter is driven by a formatting program called TROFF [4]. TROFF was designed for setting running text. It also provides all of the facilities that one needs for doing mathematics, such as arbitrary horizontal and vertical motions, line-drawing, size changing, but the syntax for describing these special operations is difficult to learn, and difficult even for experienced users to type correctly.

For this reason we decided to use TROFF as an "assembly language," by designing a language for describing mathematical expressions, and compiling it into TROFF.

3. Language Design

The fundamental principle upon which we based our language design is that the language should be easy to use by people (for example, secretaries) who know neither mathematics nor typesetting.

This principle implies several things. First, "normal" mathematical conventions about operator precedence, parentheses, and the like cannot be used, for to give special meaning to such characters means that the user has to understand what he or she is typing. Thus the language should not assume, for instance, that parentheses are always balanced, for they are not in the half-open interval $(a,b]$. Nor should it assume that that $\sqrt{a+b}$ can be replaced by $(a+b)^{1/2}$, or that $1/(1-x)$ is better written as $\frac{1}{1-x}$ (or vice versa).

Second, there should be relatively few rules, keywords, special symbols and operators, and the like. This keeps the language easy to learn and remember. Furthermore, there should be few exceptions to the rules that do exist: if something works in one situation, it should work everywhere. If a variable can have a subscript, then a subscript can have a subscript, and so on without limit.

Third, "standard" things should happen automatically. Someone who types "x=y+z+1" should get "$x=y+z+1$". Subscripts and superscripts should automatically be printed in an appropriately smaller size, with no special intervention. Fraction bars have to be made the right length and positioned at the right height. And so on. Indeed a mechanism for overriding default actions has to exist, but its application is the exception, not the rule.

We assume that the typist has a reasonable picture (a two-dimensional representation) of the desired final form, as might be handwritten by the author of a paper. We also assume that the input is typed on a computer terminal much like an ordinary typewriter. This implies an input alphabet of perhaps 100 characters, none of them special.

A secondary, but still important, goal in our design was that the system should be easy to implement, since neither of the authors had any desire to make a long-term project of it. Since our design was not firm, it was also necessary that the program be easy to change at any time.

To make the program easy to build and to change, and to guarantee regularity ("it should work everywhere"), the language is defined by a context-free grammar, described in Section 5. The compiler for the language was built using a compiler-compiler.

A priori, the grammar/compiler-compiler approach seemed the right thing to do. Our subsequent experience leads us to believe that any other course would have been folly. The original language was designed in a few days. Construction of a working system sufficient to try significant examples required perhaps a person-month. Since then, we have spent a modest amount of additional time over several years tuning, adding facilities, and occasionally changing the language as users make criticisms and suggestions.

We also decided quite early that we would let TROFF do our work for us whenever possible. TROFF is quite a powerful program, with a macro facility, text and arithmetic variables, numerical computation and testing, and conditional branching. Thus we have been able to avoid writing a lot of mundane but tricky software. For example, we store no text strings, but simply pass them on to TROFF. Thus we avoid having to write a storage management package. Furthermore, we have been able to isolate ourselves from most details of the particular device and character set currently in use. For example, we let TROFF compute the widths of all strings of characters; we need know nothing about them.

A third design goal is special to our environment. Since our program is only useful for typesetting mathematics, it is necessary that it interface cleanly with the underlying typesetting language for the benefit of users who want to set intermingled mathematics and text (the usual case). The standard mode of operation is that when a document is typed, mathematical expressions are input as part of the text, but marked by user settable delimiters. The program reads this input and treats as comments those things which are not mathematics, simply passing them through untouched. At the same time it converts the mathematical input into the necessary TROFF commands. The resulting ioutput is passed directly to TROFF where the comments and the mathematical parts both become text and/or TROFF commands.

4. The Language

We will not try to describe the language precisely here; interested readers may refer to the appendix for more details. Throughout this section, we will write expressions exactly as they are handed to the typesetting program (hereinafter called "EQN"), except that we won't show the delimiters that the user types to mark the beginning and end of the expression. The interface between EQN and TROFF is described at the end of this section.

As we said, typing x=y+z+1 should produce $x=y+z+1$, and indeed it does. Variables are made italic, operators and digits become roman, and normal spacings between letters and operators are altered slightly to give a more pleasing appearance.

Input is free-form. Spaces and new lines in the input are used by EQN to separate pieces of the input; they are not used to create space in the output. Thus

```
x   =   y
  + z + 1
```

also gives $x=y+z+1$. Free-form input is easier to type initially; subsequent editing is also easier, for an expression may be typed as many short lines.

Extra white space can be forced into the output by several characters of various sizes. A tilde "~" gives a space equal to the normal word spacing in text; a circumflex gives half this much, and a tab charcter spaces to the next tab stop.

Spaces (or tildes, etc.) also serve to delimit pieces of the input. For example, to get

$$f(t)=2\pi\int\sin(\omega t)dt$$

we write

```
f(t) = 2 pi int sin ( omega t )dt
```

Here spaces are *necessary* in the input to indicate that *sin*, *pi*, *int*, and *omega* are special, and potentially worth special treatment. EQN looks up each such string of characters in a table, and if appropriate gives it a translation. In this case, *pi* and *omega* become their greek equivalents, *int* becomes the integral sign (which must be moved down and enlarged so it looks "right"), and *sin* is made roman, following conventional mathematical practice. Parentheses, digits and operators are automatically made roman wherever found.

Fractions are specified with the keyword *over:*

```
a+b over c+d+e = 1
```

produces

$$\frac{a+b}{c+d+e}=1$$

Similarly, subscripts and superscripts are introduced by the keywords *sub* and *sup:*

$$x^2+y^2=z^2$$

is produced by

```
x sup 2 + y sup 2 = z sup 2
```

The spaces after the 2's are necessary to mark the end of the superscripts; similarly the keyword *sup* has to be marked off by spaces or some equivalent delimiter. The return to the proper baseline is automatic. Multiple levels of subscripts or superscripts are of course allowed: "x sup y sup z" is x^{y^z}. The construct "something *sub* something *sup* something" is recognized as a special case, so "x sub i sup 2" is x_i^2 instead of x_{i^2}.

More complicated expressions can now be formed with these primitives:

$$\frac{\partial^2 f}{\partial x^2}=\frac{x^2}{a^2}+\frac{y^2}{b^2}$$

is produced by

```
{partial sup 2 f} over {partial x sup 2} =
x sup 2 over a sup 2 + y sup 2 over b sup 2
```

Braces {} are used to group objects together; in this case they indicate unambiguously what goes over what on the left-hand side of the expression. The language defines the precedence of *sup* to be higher than that of *over*, so no braces are needed to get the correct association on the right side. Braces can always be used when in doubt about precedence.

The braces convention is an example of

the power of using a recursive grammar to define the language. It is part of the language that if a construct can appear in some context, then *any expression* in braces can also occur in that context.

There is a *sqrt* operator for making square roots of the appropriate size: "sqrt a+b" produces $\sqrt{a+b}$, and

```
x = {−b +− sqrt{b sup 2 −4ac}} over 2a
```

is

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

Since large radicals look poor on our typesetter, *sqrt* is not useful for tall expressions.

Limits on summations, integrals and similar constructions are specified with the keywords *from* and *to*. To get

$$\sum_{i=0}^{\infty} x_i \to 0$$

we need only type

```
sum from i=0 to inf x sub i −> 0
```

Centering and making the Σ big enough and the limits smaller are all automatic. The *from* and *to* parts are both optional, and the central part (e.g., the Σ) can in fact be anything:

```
lim from {x −> pi /2} ( tan~x) = inf
```

is

$$\lim_{x\to\pi/2} (\tan x)=\infty$$

Again, the braces indicate just what goes into the *from* part.

There is a facility for making braces, brackets, parentheses, and vertical bars of the right height, using the keywords *left* and *right:*

```
left [ x+y over 2a right ]~=~1
```

makes

$$\left[\frac{x+y}{2a}\right]=1$$

A *left* need not have a corresponding *right*, as we shall see in the next example. Any characters may follow *left* and *right*, but generally only various parentheses and bars are meaningful.

Big brackets, etc., are often used with another facility, called *piles*, which make vertical piles of objects. For example, to get

$$sign\,(x) \equiv \left\{ \begin{array}{rl} 1 & \text{if } x>0 \\ 0 & \text{if } x=0 \\ -1 & \text{if } x<0 \end{array} \right.$$

we can type

```
sign (x) ~==~ left {
   rpile {1 above 0 above −1}
   ~~lpile {if above if above if}
   ~~lpile {x>0 above x=0 above x<0}
```

The construction "left {" makes a left brace big enough to enclose the "rpile {...}", which is a right-justified pile of "above ... above ...". "lpile" makes a left-justified pile. There are also centered piles. Because of the recursive language definition, a pile can contain any number of elements; any element of a pile can of course contain piles.

Although EQN makes a valiant attempt to use the right sizes and fonts, there are times when the default assumptions are simply not what is wanted. For instance the italic *sign* in the previous example would conventionally be in roman. Slides and transparencies often require larger characters than normal text. Thus we also provide size and font changing commands: "size 12 bold {A~x~=~y}" will produce $\mathbf{A\ x = y}$. *Size* is followed by a number representing a character size in points. (One point is 1/72 inch; this paper is set in 9 point type.)

If necessary, an input string can be quoted in "...", which turns off grammatical significance, and any font or spacing changes that might otherwise be done on it. Thus we can say

```
lim~ roman "sup" ~x sub n = 0
```

to ensure that the supremum doesn't become a superscript:

$$\lim \mathrm{sup}\; x_n=0$$

Diacritical marks, long a problem in traditional typesetting, are straightforward:

$$\underset{\cdot}{x}+\hat{x}+\tilde{y}+\hat{X}+\ddot{Y}=\overline{z+Z}$$

is made by typing

```
x dot under + x hat + y tilde
+ X hat + Y dotdot = z+Z bar
```

There are also facilities for globally changing default sizes and fonts, for example for making viewgraphs or for setting chemical equations. The language allows for matrices, and for lining up equations at the same horizontal position.

Finally, there is a definition facility, so a user can say

```
define name "..."
```

at any time in the document; henceforth, any occurrence of the token "name" in an expression will be expanded into whatever was inside the double quotes in its definition. This lets users tailor the language to their own specifications, for it is quite possible to redefine keywords like *sup* or *over*. Section 6 shows an example of definitions.

The EQN preprocessor reads intermixed text and equations, and passes its output to TROFF. Since TROFF uses lines beginning with a period as control words (e.g., ".ce" means "center the next output line"), EQN uses the sequence ".EQ" to mark the beginning of an equation and ".EN" to mark the end. The ".EQ" and ".EN" are passed through to TROFF untouched, so they can also be used by a knowledgeable user to center equations, number them automatically, etc. By default, however, ".EQ" and ".EN" are simply ignored by TROFF, so by default equations are printed in-line.

".EQ" and ".EN" can be supplemented by TROFF commands as desired; for example, a centered display equation can be produced with the input:

```
.ce
.EQ
x sub i = y sub i ...
.EN
```

Since it is tedious to type ".EQ" and ".EN" around very short expressions (single letters, for instance), the user can also define two characters to serve as the left and right delimiters of expressions. These characters are recognized anywhere in subsequent text. For example if the left and right delimiters have both been set to "#", the input:

```
Let #x sub i#, #y# and #alpha# be positive
```

produces:

Let x_i, y and α be positive

Running a preprocessor is strikingly easy on UNIX. To typeset text stored in file "f", one issues the command:

```
eqn f | troff
```

The vertical bar connects the output of one process (EQN) to the input of another (TROFF).

5. Language Theory

The basic structure of the language is not a particularly original one. Equations are pictured as a set of "boxes," pieced together in various ways. For example, something with a subscript is just a box followed by another box moved downward and shrunk by an appropriate amount. A fraction is just a box centered above another box, at the right altitude, with a line of correct length drawn between them.

The grammar for the language is shown below. For purposes of exposition, we have collapsed some productions. In the original grammar, there are about 70 productions, but many of these are simple ones used only to guarantee that some keyword is recognized early enough in the parsing process. Symbols in capital letters are terminal symbols; lower case symbols are non-terminals, i.e., syntactic categories. The vertical bar | indicates an alternative; the brackets [ ] indicate optional material. A TEXT is a string of non-blank characters or any string inside double quotes; the other terminal symbols represent literal occurrences of the corresponding keyword.

```
eqn  : box | eqn box

box  : text
     | { eqn }
     | box OVER box
     | SQRT box
     | box SUB box | box SUP box
     | [ L | C | R ]PILE { list }
     | LEFT text eqn [ RIGHT text ]
     | box [ FROM box ] [ TO box ]
     | SIZE text box
     | [ROMAN | BOLD | ITALIC] box
     | box [HAT | BAR | DOT | DOTDOT | TILDE]
     | DEFINE text text

list  : eqn | list ABOVE eqn

text  : TEXT
```

The grammar makes it obvious why there are few exceptions. For example, the observation that something can be replaced by a more complicated something in braces is implicit in the productions:

```
eqn  : box | eqn box
box  : text | { eqn }
```

Anywhere a single character could be used, *any* legal construction can be used.

Clearly, our grammar is highly ambiguous. What, for instance, do we do with the input

```
a over b over c  ?
```

Is it

```
{a over b} over c
```

or is it

```
a over {b over c}  ?
```

To answer questions like this, the grammar is supplemented with a small set of rules that describe the precedence and associativity of operators. In particular, we specify (more or less arbitrarily) that *over* associates to the left, so the first alternative above is the one chosen. On the other hand, *sub* and *sup* bind to the right, because this is closer to standard mathematical practice. That is, we assume x^{a^b} is $x^{(a^b)}$, not $(x^a)^b$.

The precedence rules resolve the ambiguity in a construction like

```
a sup 2 over b
```

We define *sup* to have a higher precedence than *over*, so this construction is parsed as $\frac{a^2}{b}$ instead of $a^{\frac{2}{b}}$.

Naturally, a user can always force a particular parsing by placing braces around expressions.

The ambiguous grammar approach seems to be quite useful. The grammar we use is small enough to be easily understood, for it contains none of the productions that would be normally used for resolving ambiguity. Instead the supplemental information about precedence and associativity (also small enough to be understood) provides the compiler-compiler with the information it needs to make a fast, deterministic parser for the specific language we want. When the language is supplemented by the disambiguating rules, it is in fact LR(1) and thus easy to parse[5].

The output code is generated as the input is scanned. Any time a production of the grammar is recognized, (potentially) some TROFF commands are output. For example, when the lexical analyzer reports that it has found a TEXT (i.e., a string of contiguous characters), we have recognized the production:

```
text    : TEXT
```

The translation of this is simple. We generate a local name for the string, then hand the name and the string to TROFF, and let TROFF perform the storage management. All we save is the name of the string, its height, and its baseline.

As another example, the translation associated with the production

```
box    : box OVER box
```

is:

```
Width of output box =
  slightly more than largest input width
Height of output box =
  slightly more than sum of input heights
Base of output box =
  slightly more than height of bottom input box
String describing output box =
  move down;
  move right enough to center bottom box;
  draw bottom box (i.e., copy string for bottom box);
  move up; move left enough to center top box;
  draw top box (i.e., copy string for top box);
  move down and left; draw line full width;
  return to proper base line.
```

Most of the other productions have equally simple semantic actions. Picturing the output as a set of properly placed boxes makes the right sequence of positioning commands quite obvious. The main difficulty is in finding the right numbers to use for esthetically pleasing positioning.

With a grammar, it is usually clear how to extend the language. For instance, one of our users suggested a TENSOR operator, to make constructions like

$${}^{l}_{m}\overset{k}{\underset{n}{\mathrm{T}}}{}^{j}_{i}$$

Grammatically, this is easy: it is sufficient to add a production like

```
box    : TENSOR { list }
```

Semantically, we need only juggle the boxes to the right places.

6. Experience

There are really three aspects of interest—how well EQN sets mathematics, how well it satisfies its goal of being "easy to use," and how easy it was to build.

The first question is easily addressed. This entire paper has been set by the program. Readers can judge for themselves whether it is good enough for their purposes. One of our users commented that although the output is not as good as the best hand-set material, it is still better than average, and much better than the worst. In any case, who cares? Printed books cannot compete with the birds and flowers of illuminated manuscripts on esthetic grounds, either, but they have some clear economic advantages.

Some of the deficiencies in the output could be cleaned up with more work on our part. For example, we sometimes leave too much space between a roman letter and an italic one. If we were willing to keep track of the fonts involved, we could do this better more of the time.

Some other weaknesses are inherent in our output device. It is hard, for instance, to draw a line of an arbitrary length without getting a perceptible overstrike at one end.

As to ease of use, at the time of writing, the system has been used by two distinct groups. One user population consists of mathematicians, chemists, physicists, and computer scientists. Their typical reaction has been something like:

(1) It's easy to write, although I make the following mistakes...

(2) How do I do...?

(3) It botches the following things.... Why don't you fix them?

(4) You really need the following features...

The learning time is short. A few minutes gives the general flavor, and typing a page or two of a paper generally uncovers most of the misconceptions about how it works.

The second user group is much larger, the secretaries and mathematical typists who were the original target of the system. They tend to be enthusiastic converts. They find the language easy to learn (most are largely self-taught), and have little trouble producing the output they want. They are of course less critical of the esthetics of their output than users trained in mathematics. After a transition period, most find using a computer more interesting than a regular typewriter.

The main difficulty that users have seems to be remembering that a blank is a delimiter; even experienced users use blanks where they shouldn't and omit them when they are needed. A common instance is typing

```
f(x sub i)
```

which produces

$$f(x_{i)}$$

instead of

$$f(x_i)$$

Since the EQN language knows no mathematics, it cannot deduce that the right parenthesis is not part of the subscript.

The language is somewhat prolix, but this doesn't seem excessive considering how much is being done, and it is certainly more compact than the corresponding TROFF commands. For example, here is the source for the continued fraction expression in Section 1 of this paper:

```
a sub 0 + b sub 1 over
  {a sub 1 + b sub 2 over
    {a sub 2 + b sub 3 over
      {a sub 3 + ... }}}
```

This is the input for the large integral of Section 1; notice the use of definitions:

```
define emx "{e sup mx}"
define mab "{m sqrt ab}"
define sa "{sqrt a}"
define sb "{sqrt b}"
int dx over {a emx - be sup -mx} ~=~
left { lpile {
     1 over {2 mab} ~log~
          {sa emx - sb} over {sa emx + sb}
   above
     1 over mab ~ tanh sup -1 ( sa over sb emx )
   above
     -1 over mab ~ coth sup -1 ( sa over sb emx )
}
```

As to ease of construction, we have already mentioned that there are really only a few person-months invested. Much of this time has gone into two things—fine-tuning (what is the most esthetically pleasing space to use between the numerator and denominator of a fraction?), and changing things found deficient by our users (shouldn't a tilde be a delimiter?).

The program consists of a number of small, essentially unconnected modules for code generation, a simple lexical analyzer, a canned parser which we did not have to write, and some miscellany associated with input files and the macro facility. The program is now about 1600 lines of C [6], a high-level language reminiscent of BCPL. About 20 percent of these lines are "print" statements, generating the output code.

The semantic routines that generate the actual TROFF commands can be changed to accommodate other formatting languages and devices. For example, in less than 24 hours, one of us changed the entire semantic package to drive NROFF, a variant of TROFF, for typesetting mathematics on teletypewriter devices capable of reverse line motions. Since many potential users do not have access to a typesetter, but still have to type mathematics, this provides a way to get a typed version of the final output which is close enough for debugging purposes, and sometimes even for ultimate use.

7. Conclusions

We think we have shown that it is possible to do acceptably good typesetting of mathematics on a phototypesetter, with an input language that is easy to learn and use and that satisfies many users' demands. Such a package can be implemented in short order, given a compiler-compiler and a decent typesetting program underneath.

Defining a language, and building a compiler for it with a compiler-compiler seems like the only sensible way to do business. Our experience with the use of a grammar and a compiler-compiler has been uniformly favorable. If we had written everything into code directly, we would have been locked into our original design. Furthermore, we would have never been sure where the exceptions and special cases were. But because we have a grammar, we can change our minds readily and still be reasonably sure that if a construction works in one place it will work everywhere.

Acknowledgements

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to modify TROFF to make our task easier and for his continuous assistance during the development of our program. We are also grateful to A. V. Aho for help with language theory, to S. C. Johnson for aid with the compiler-compiler, and to our early users A. V. Aho, S. I. Feldman, S. C. Johnson, R. W. Hamming, and M. D. McIlroy for their constructive criticisms.

References

[1] *A Manual of Style.* 12th Edition. University of Chicago Press, 1969. p 295.

[2] *Model C/A/T Phototypesetter.* Graphic Systems, Inc., Hudson, N. H.

[3] Ritchie, D. M., and Thompson, K. L., "The UNIX time-sharing system." *Comm. ACM 17,* 7 (July 1974), 365-375.

[4] Ossanna, J. F., TROFF User's Manual. Bell Laboratories Computing Science Technical Report 54, 1977.

[5] Aho, A. V., and Johnson, S. C., "LR Parsing." *Comp. Surv. 6,* 2 (June 1974), 99-124.

[6] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Prentice-Hall, Inc., 1978.

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This is the user's guide for a system for typesetting mathematics, using the phototypesetters on the UNIX† and GCOS operating systems.

Mathematical expressions are described in a language designed to be easy to use by people who know neither mathematics nor typesetting. Enough of the language to set in-line expressions like $\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$ or display equations like

$$G(z) = e^{\ln G(z)} = \exp \left(\sum_{k \geq 1} \frac{S_k z^k}{k} \right) = \prod_{k \geq 1} e^{S_k z^k / k}$$

$$= \left(1 + S_1 z + \frac{S_1^2 z^2}{2!} + \cdots \right) \left(1 + \frac{S_2 z^2}{2} + \frac{S_2^2 z^4}{2^2 \cdot 2!} + \cdots \right) \cdots$$

$$= \sum_{m \geq 0} \left(\sum_{\substack{k_1, k_2, \ldots, k_m \geq 0 \\ k_1 + 2k_2 + \cdots + mk_m = m}} \frac{S_1^{k_1}}{1^{k_1} k_1!} \, \frac{S_2^{k_2}}{2^{k_2} k_2!} \cdots \frac{S_m^{k_m}}{m^{k_m} k_m!} \right) z^m$$

can be learned in an hour or so.

The language interfaces directly with the phototypesetting language TROFF, so mathematical expressions can be embedded in the running text of a manuscript, and the entire document produced in one process. This user's guide is an example of its output.

The same language may be used with the UNIX formatter NROFF to set mathematical expressions on DASI and GSI terminals and Model 37 teletypes.

August 15, 1978

†UNIX is a Trademark of Bell Laboratories.

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

EQN is a program for typesetting mathematics on the Graphics Systems phototypesetters on UNIX and GCOS. The EQN language was designed to be easy to use by people who know neither mathematics nor typesetting. Thus EQN knows relatively little about mathematics. In particular, mathematical symbols like +, −, ×, parentheses, and so on have no special meanings. EQN is quite happy to set garbage (but it will look good).

EQN works as a preprocessor for the typesetter formatter, TROFF[1], so the normal mode of operation is to prepare a document with both mathematics and ordinary text interspersed, and let EQN set the mathematics while TROFF does the body of the text.

On UNIX, EQN will also produce mathematics on DASI and GSI terminals and on Model 37 teletypes. The input is identical, but you have to use the programs NEQN and NROFF instead of EQN and TROFF. Of course, some things won't look as good because terminals don't provide the variety of characters, sizes and fonts that a typesetter does, but the output is usually adequate for proofreading.

To use EQN on UNIX,

```
eqn files | troff
```

GCOS use is discussed in section 26.

2. Displayed Equations

To tell EQN where a mathematical expression begins and ends, we mark it with lines beginning .EQ and .EN. Thus if you type the lines

```
.EQ
x=y+z
.EN
```

your output will look like

$$x=y+z$$

The .EQ and .EN are copied through untouched; they are not otherwise processed by EQN. This means that you have to take care of things like centering, numbering, and so on yourself. The most common way is to use the TROFF and NROFF macro package package '−ms' developed by M. E. Lesk[3], which allows you to center, indent, left-justify and number equations.

With the '−ms' package, equations are centered by default. To left-justify an equation, use .EQ L instead of .EQ. To indent it, use .EQ I. Any of these can be followed by an arbitrary 'equation number' which will be placed at the right margin. For example, the input

```
.EQ I (3.1a)
x = f(y/2) + y/2
.EN
```

produces the output

$$x=f(y/2)+y/2 \qquad (3.1a)$$

There is also a shorthand notation so in-line expressions like π_i^2 can be entered without .EQ and .EN. We will talk about it in section 19.

3. Input spaces

Spaces and newlines within an expression are thrown away by EQN. (Normal text is left absolutely alone.) Thus between .EQ and .EN,

```
x=y+z
```

and

```
x = y + z
```

and

```
x   =   y
  + z
```

and so on all produce the same output

$$x=y+z$$

You should use spaces and newlines freely to make your input equations readable and easy to edit. In particular, very long lines are a bad idea, since they are often hard to fix if you make a mistake.

4. Output spaces

To force extra spaces into the *output,* use a tilde "~" for each space you want:

```
x~=~y~+~z
```

gives

$$x ~ = ~ y ~ + ~ z$$

You can also use a circumflex "^", which gives a space half the width of a tilde. It is mainly useful for fine-tuning. Tabs may also be used to position pieces of an expression, but the tab stops must be set by TROFF commands.

5. Symbols, Special Names, Greek

EQN knows some mathematical symbols, some mathematical names, and the Greek alphabet. For example,

```
x=2 pi int sin ( omega t)dt
```

produces

$$x=2\pi\int\sin(\omega t)\,dt$$

Here the spaces in the input are necessary to tell EQN that *int, pi, sin* and *omega* are separate entities that should get special treatment. The *sin,* digit 2, and parentheses are set in roman type instead of italic; *pi* and *omega* are made Greek; and *int* becomes the integral sign.

When in doubt, leave spaces around separate parts of the input. A *very* common error is to type *f(pi)* without leaving spaces on both sides of the *pi.* As a result, EQN does not recognize *pi* as a special word, and it appears as $f(pi)$ instead of $f(\pi)$.

A complete list of EQN names appears in section 23. Knowledgeable users can also use TROFF four-character names for anything EQN doesn't know about, like *\(bs* for the Bell System sign.

6. Spaces, Again

The only way EQN can deduce that some sequence of letters might be special is if that sequence is separated from the letters on either side of it. This can be done by surrounding a special word by ordinary spaces (or tabs or newlines), as we did in the previous section.

You can also make special words stand out by surrounding them with tildes or circumflexes:

```
x~=~2~pi~int~sin~(~omega~t~)~dt
```

is much the same as the last example, except that the tildes not only separate the magic words like *sin, omega,* and so on, but also add extra spaces, one space per tilde:

$$x ~ = ~ 2 ~ \pi ~ \int ~ \sin ~ (~ \omega ~ t ~) ~ dt$$

Special words can also be separated by braces { } and double quotes "...", which have special meanings that we will see soon.

7. Subscripts and Superscripts

Subscripts and superscripts are obtained with the words *sub* and *sup.*

```
x sup 2 + y sub k
```

gives

$$x^2+y_k$$

EQN takes care of all the size changes and vertical motions needed to make the output look right. The words *sub* and *sup* must be surrounded by spaces; *x sub2* will give you *xsub2* instead of x_2. Furthermore, don't forget to leave a space (or a tilde, etc.) to mark the end of a subscript or superscript. A common error is to say something like

```
y = (x sup 2)+1
```

which causes

$$y=(x^{2)+1}$$

instead of the intended

$$y=(x^2)+1$$

Subscripted subscripts and superscripted superscripts also work:

```
x sub i sub 1
```

is

$$x_{i_1}$$

A subscript and superscript on the same thing are printed one above the other if the subscript comes *first:*

```
x sub i sup 2
```

is

$$x_i^2$$

Other than this special case, *sub* and *sup* group to the right, so *x sup y sub z* means x^{y_z}, not $x^y{}_z$.

8. Braces for Grouping

Normally, the end of a subscript or superscript is marked simply by a blank (or tab or tilde, etc.) What if the subscript or superscript is something that has to be typed with blanks in it? In that case, you can use the braces { and } to mark the beginning and end of the subscript or superscript:

```
e sup {i omega t}
```

is

$$e^{i\omega t}$$

Rule: Braces can *always* be used to force EQN to treat something as a unit, or just to make your intent perfectly clear. Thus:

```
x sub {i sub 1} sup 2
```

is

$$x_{i_1}^2$$

with braces, but

```
x sub i sub 1 sup 2
```

is

$$x_{i_1^2}$$

which is rather different.

Braces can occur within braces if necessary:

```
e sup {i pi sup {rho +1}}
```

is

$$e^{i\pi^{\rho+1}}$$

The general rule is that anywhere you could use some single thing like *x*, you can use an arbitrarily complicated thing if you enclose it in braces. EQN will look after all the details of positioning it and making it the right size.

In all cases, make sure you have the right number of braces. Leaving one out or adding an extra will cause EQN to complain bitterly.

Occasionally you will have to print braces. To do this, enclose them in double quotes, like "{". Quoting is discussed in more detail in section 14.

9. Fractions

To make a fraction, use the word *over:*

```
a+b over 2c =1
```

gives

$$\frac{a+b}{2c}=1$$

The line is made the right length and positioned automatically. Braces can be used to make clear what goes over what:

```
{alpha + beta} over {sin (x)}
```

is

$$\frac{\alpha+\beta}{\sin(x)}$$

What happens when there is both an *over* and a *sup* in the same expression? In such an apparently ambiguous case, EQN does the *sup* before the *over,* so

```
-b sup 2 over pi
```

is $\frac{-b^2}{\pi}$ instead of $-b^{\frac{2}{\pi}}$ The rules which decide which operation is done first in cases like this are summarized in section 23. When in doubt, however, *use braces* to make clear what goes with what.

10. Square Roots

To draw a square root, use *sqrt:*

```
sqrt a+b + 1 over sqrt {ax sup 2 +bx+c}
```

is

$$\sqrt{a+b}+\frac{1}{\sqrt{ax^2+bx+c}}$$

Warning — square roots of tall quantities look lousy, because a root-sign big enough to cover the quantity is too dark and heavy:

    sqrt {a sup 2 over b sub 2}

is

$$\sqrt{\frac{a^2}{b_2}}$$

Big square roots are generally better written as something to the power ½:

$$(a^2/b_2)^{1/2}$$

which is

    (a sup 2 /b sub 2) sup half

11. Summation, Integral, Etc.

Summations, integrals, and similar constructions are easy:

    sum from i=0 to {i= inf} x sup i

produces

$$\sum_{i=0}^{i=\infty} x^i$$

Notice that we used braces to indicate where the upper part $i=\infty$ begins and ends. No braces were necessary for the lower part $i=0$, because it contained no blanks. The braces will never hurt, and if the *from* and *to* parts contain any blanks, you must use braces around them.

The *from* and *to* parts are both optional, but if both are used, they have to occur in that order.

Other useful characters can replace the *sum* in our example:

    int prod union inter

become, respectively,

$$\int \quad \prod \quad \bigcup \quad \bigcap$$

Since the thing before the *from* can be anything, even something in braces, *from-to* can often be used in unexpected ways:

    lim from {n —> inf} x sub n =0

is

$$\lim_{n \to \infty} x_n = 0$$

12. Size and Font Changes

By default, equations are set in 10-point type (the same size as this guide), with standard mathematical conventions to determine what characters are in roman and what in italic. Although EQN makes a valiant attempt to use esthetically pleasing sizes and fonts, it is not perfect. To change sizes and fonts, use *size n* and *roman, italic, bold* and *fat*. Like *sub* and *sup*, size and font changes affect only the thing that follows them, and revert to the normal situation at the end of it. Thus

    bold x y

is

$$\mathbf{x}y$$

and

    size 14 bold x = y +
      size 14 {alpha + beta}

gives

$$\mathbf{x}=y+\alpha+\beta$$

As always, you can use braces if you want to affect something more complicated than a single letter. For example, you can change the size of an entire equation by

    size 12 { ... }

Legal sizes which may follow *size* are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, 36. You can also change the size *by* a given amount; for example, you can say *size +2* to make the size two points bigger, or *size −3* to make it three points smaller. This has the advantage that you don't have to know what the current size is.

If you are using fonts other than roman, italic and bold, you can say *font X* where *X* is a one character TROFF name or number for the font. Since EQN is tuned for roman, italic and bold, other fonts may not give quite as good an appearance.

The *fat* operation takes the current font and widens it by overstriking: *fat grad* is ∇ and *fat {x sub i}* is x_i.

If an entire document is to be in a non-standard size or font, it is a severe nuisance to have to write out a size and font change for each equation. Accordingly, you can set a "global" size or font which thereafter affects all equations. At the beginning of any equation, you might say, for instance,

```
.EQ
gsize 16
gfont R
...
.EN
```

to set the size to 16 and the font to roman thereafter. In place of R, you can use any of the TROFF font names. The size after *gsize* can be a relative change with + or −.

Generally, *gsize* and *gfont* will appear at the beginning of a document but they can also appear thoughout a document: the global font and size can be changed as often as needed. For example, in a footnote‡ you will typically want the size of equations to match the size of the footnote text, which is two points smaller than the main text. Don't forget to reset the global size at the end of the footnote.

13. Diacritical Marks

To get funny marks on top of letters, there are several words:

x dot	$\dot{x}$
x dotdot	$\ddot{x}$
x hat	$\hat{x}$
x tilde	$\tilde{x}$
x vec	$\vec{x}$
x dyad	$\overleftrightarrow{x}$
x bar	$\bar{x}$
x under	$\underline{x}$

The diacritical mark is placed at the right height. The *bar* and *under* are made the right length for the entire construct, as in $\overline{x+y+z}$; other marks are centered.

14. Quoted Text

Any input entirely within quotes ("...") is not subject to any of the font changes and spacing adjustments normally done by the equation setter. This provides a way to do your own spacing and adjusting if needed:

italic "sin(x)" + sin (x)

is

sin(x)+sin(*x*)

Quotes are also used to get braces and other EQN keywords printed:

"{ size alpha }"

is

{ *size alpha* }

and

roman "{ size alpha }"

is

{ size alpha }

The construction "" is often used as a place-holder when grammatically EQN needs something, but you don't actually want anything in your output. For example, to make ${}^2\mathrm{He}$, you can't just type *sup 2 roman He* because a *sup* has to be a superscript *on* something. Thus you must say

"" sup 2 roman He

To get a literal quote use "\"". TROFF characters like \(bs can appear unquoted, but more complicated things like horizontal and vertical motions with \h and \v should always be quoted. (If you've never heard of \h and \v, ignore this section.)

15. Lining Up Equations

Sometimes it's necessary to line up a series of equations at some horizontal position, often at an equals sign. This is done with two operations called *mark* and *lineup*.

The word *mark* may appear once at any place in an equation. It remembers the horizontal position where it appeared. Successive equations can contain one occurrence of the word *lineup*. The place where *lineup* appears is made to line up with the place marked by the previous *mark* if at all possible. Thus, for example, you can say

‡Like this one, in which we have a few random expressions like x_i and π^2. The sizes for these were set by the command *gsize −2*.

```
.EQ I
x+y mark = z
.EN
.EQ I
x lineup = 1
.EN
```

to produce

$$x+y=z$$

$$x=1$$

For reasons too complicated to talk about, when you use EQN and '-ms', use either .EQ I or .EQ L. mark and *lineup* don't work with centered equations. Also bear in mind that *mark* doesn't look ahead;

```
x mark =1
...
x+y lineup =z
```

isn't going to work, because there isn't room for the $x+y$ part after the *mark* remembers where the x is.

16. Big Brackets, Etc.

To get big brackets [ ], braces { }, parentheses ( ), and bars || around things, use the *left* and *right* commands:

```
left { a over b + 1 right }
~=~ left ( c over d right )
 + left [ e right ]
```

is

$$\left\{ \frac{a}{b}+1 \right\} = \left(\frac{c}{d} \right) + \left[e \right]$$

The resulting brackets are made big enough to cover whatever they enclose. Other characters can be used besides these, but the are not likely to look very good. One exception is the *floor* and *ceiling* characters:

```
left floor x over y right floor
<= left ceiling a over b right ceiling
```

produces

$$\left\lfloor \frac{x}{y} \right\rfloor \leq \left\lceil \frac{a}{b} \right\rceil$$

Several warnings about brackets are in order. First, braces are typically bigger than brackets and parentheses, because they are made up of three, five, seven, etc., pieces, while brackets can be made up of two, three, etc. Second, big left and right parentheses often look poor, because the character set is poorly designed.

The *right* part may be omitted: a "left something" need not have a corresponding "right something". If the *right* part is omitted, put braces around the thing you want the left bracket to encompass. Otherwise, the resulting brackets may be too large.

If you want to omit the *left* part, things are more complicated, because technically you can't have a *right* without a corresponding *left*. Instead you have to say

```
left "" ..... right )
```

for example. The *left* "" means a "left nothing". This satisfies the rules without hurting your output.

17. Piles

There is a general facility for making vertical piles of things; it comes in several flavors. For example:

```
A ~=~ left [
  pile { a above b above c }
  ~~ pile { x above y above z }
right ]
```

will make

$$A = \left[\begin{matrix} a & x \\ b & y \\ c & z \end{matrix} \right]$$

The elements of the pile (there can be as many as you want) are centered one above another, at the right height for most purposes. The keyword *above* is used to separate the pieces; braces are used around the entire list. The elements of a pile can be as complicated as needed, even containing more piles.

Three other forms of pile exist: *lpile* makes a pile with the elements left-justified; *rpile* makes a right-justified pile; and *cpile* makes a centered pile, just like *pile*. The vertical spacing between the pieces is somewhat larger for *l-*, *r-* and *cpiles* than it is for ordinary piles.

```
roman sign (x)~=~
left {
   lpile {1 above 0 above -1}
   ~~ lpile
   {if~x>0 above if~x=0 above if~x<0}
```

makes

$$\mathrm{sign}(x) = \left\{ \begin{array}{ll} 1 & \text{if } x>0 \\ 0 & \text{if } x=0 \\ -1 & \text{if } x<0 \end{array} \right.$$

Notice the left brace without a matching right one.

18. Matrices

It is also possible to make matrices. For example, to make a neat array like

$$\begin{matrix} x_i & x^2 \\ y_i & y^2 \end{matrix}$$

you have to type

```
    matrix {
      ccol { x sub i above y sub i }
      ccol { x sup 2 above y sup 2 }
    }
```

This produces a matrix with two centered columns. The elements of the columns are then listed just as for a pile, each element separated by the word *above*. You can also use *lcol* or *rcol* to left or right adjust columns. Each column can be separately adjusted, and there can be as many columns as you like.

The reason for using a matrix instead of two adjacent piles, by the way, is that if the elements of the piles don't all have the same height, they won't line up properly. A matrix forces them to line up, because it looks at the entire structure before deciding what spacing to use.

A word of warning about matrices — *each column must have the same number of elements in it.* The world will end if you get this wrong.

19. Shorthand for In-line Equations

In a mathematical document, it is necessary to follow mathematical conventions not just in display equations, but also in the body of the text, for example by making variable names like x italic. Although this could be done by surrounding the appropriate parts with .EQ and .EN, the continual repetition of .EQ and .EN is a nuisance. Furthermore, with '−ms', .EQ and .EN imply a displayed equation.

EQN provides a shorthand for short in-line expressions. You can define two characters to mark the left and right ends of an in-line equation, and then type expressions right in the middle of text lines. To set both the left and right characters to dollar signs, for example, add to the beginning of your document the three lines

```
    .EQ
    delim $$
    .EN
```

Having done this, you can then say things like

> Let $alpha sub i$ be the primary variable, and let $beta$ be zero. Then we can show that $x sub 1$ is $>=0$.

This works as you might expect — spaces, newlines, and so on are significant in the text, but not in the equation part itself. Multiple equations can occur in a single input line.

Enough room is left before and after a line that contains in-line expressions that something like $\sum_{i=1}^{n} x_i$ does not interfere with the lines surrounding it.

To turn off the delimiters,

```
    .EQ
    delim off
    .EN
```

Warning: don't use braces, tildes, circumflexes, or double quotes as delimiters — chaos will result.

20. Definitions

EQN provides a facility so you can give a frequently-used string of characters a name, and thereafter just type the name instead of the whole string. For example, if the sequence

```
    x sub i sub 1 + y sub i sub 1
```

appears repeatedly throughout a paper, you can save re-typing it each time by defining it like this:

```
  define xy 'x sub i sub 1 + y sub i sub 1'
```

This makes *xy* a shorthand for whatever characters occur between the single quotes in the definition. You can use any character instead of quote to mark the ends of the definition, so long as it doesn't appear inside the definition.

Now you can use *xy* like this:

```
.EQ
f(x) = xy ...
.EN
```

and so on. Each occurrence of *xy* will expand into what it was defined as. Be careful to leave spaces or their equivalent around the name when you actually use it, so EQN will be able to identify it as special.

There are several things to watch out for. First, although definitions can use previous definitions, as in

```
.EQ
define xi  ' x sub i '
define xi1  ' xi sub 1 '
.EN
```

don't define something in terms of itself A favorite error is to say

```
define X  ' roman X '
```

This is a guaranteed disaster, since X *is* now defined in terms of itself. If you say

```
define X  ' roman "X" '
```

however, the quotes protect the second X, and everything works fine.

EQN keywords can be redefined. You can make / mean *over* by saying

```
define /  ' over '
```

or redefine *over* as / with

```
define over  ' / '
```

If you need different things to print on a terminal and on the typesetter, it is sometimes worth defining a symbol differently in NEQN and EQN. This can be done with *ndefine* and *tdefine*. A definition made with *ndefine* only takes effect if you are running NEQN; if you use *tdefine*, the definition only applies for EQN. Names defined with plain *define* apply to both EQN and NEQN.

21. Local Motions

Although EQN tries to get most things at the right place on the paper, it isn't perfect, and occasionally you will need to tune the output to make it just right. Small extra horizontal spaces can be obtained with tilde and circumflex. You can also say *back n* and *fwd n* to move small amounts horizontally. *n* is how far to move in 1/100's of an em (an em is about the width of the letter 'm'.) Thus *back 50* moves back about half the width of an m. Similarly you can move things up or down with *up n* and *down n*. As with *sub* or *sup*, the local motions affect the next thing in the input, and this can be something arbitrarily complicated if it is enclosed in braces.

22. A Large Example

Here is the complete source for the three display equations in the abstract of this guide.

```
.EQ I
G(z)~mark =~ e sup { ln ~ G(z) }
~=~ exp left (
sum from k>=1 {S sub k z sup k} over k right )
~=~  prod from k>=1 e sup {S sub k z sup k /k}
.EN
.EQ I
lineup = left ( 1 + S sub 1 z +
{ S sub 1 sup 2 z sup 2 } over 2! + ... right )
left ( 1+ { S sub 2 z sup 2 } over 2
+ { S sub 2 sup 2 z sup 4 } over { 2 sup 2 cdot 2! }
+ ... right ) ...
.EN
.EQ I
lineup =  sum from m>=0 left (
sum from
pile { k sub 1 ,k sub 2 ,..., k sub m  >=0
above
k sub 1 +2k sub 2 + ... +mk sub m =m}
{ S sub 1 sup {k sub 1} } over {1 sup k sub 1 k sub 1 ! } ~
{ S sub 2 sup {k sub 2} } over {2 sup k sub 2 k sub 2 ! } ~
...
{ S sub m sup {k sub m} } over {m sup k sub m k sub m ! }
right ) z sup m
.EN
```

23. Keywords, Precedences, Etc.

If you don't use braces, EQN will do operations in the order shown in this list.

dyad vec under bar tilde hat dot dotdot
fwd back down up
fat roman italic bold size
sub sup sqrt over
from to

These operations group to the left:

over sqrt left right

All others group to the right.

Digits, parentheses, brackets, punctuation marks, and these mathematical words are converted to Roman font when encountered:

sin cos tan sinh cosh tanh arc
max min lim log ln exp
Re Im and if for det

These character sequences are recognized and translated as shown.

>=	$\geq$
<=	$\leq$
==	$\equiv$
!=	$\neq$
+-	$\pm$
->	$\rightarrow$
<-	$\leftarrow$
<<	$<<$
>>	$>>$
inf	∞
partial	∂
half	½
prime	$'$
approx	$\approx$
nothing	
cdot	$\cdot$
times	$\times$
del	∇
grad	∇
...	$\cdots$
,...,	$,\ldots,$
sum	$\sum$
int	$\int$
prod	$\prod$
union	$\bigcup$
inter	$\bigcap$

To obtain Greek letters, simply spell them out in whatever case you want:

DELTA	Δ	iota	ι
GAMMA	Γ	kappa	κ
LAMBDA	Λ	lambda	λ
OMEGA	Ω	mu	μ
PHI	Φ	nu	ν
PI	Π	omega	ω
PSI	Ψ	omicron	o
SIGMA	Σ	phi	ϕ
THETA	Θ	pi	π
UPSILON	Υ	psi	ψ
XI	Ξ	rho	ρ
alpha	α	sigma	σ
beta	β	tau	τ
chi	χ	theta	θ
delta	δ	upsilon	υ
epsilon	ϵ	xi	ξ
eta	η	zeta	ζ
gamma	γ		

These are all the words known to EQN (except for characters with names), together with the section where they are discussed.

above	17, 18	lpile	17
back	21	mark	15
bar	13	matrix	18
bold	12	ndefine	20
ccol	18	over	9
col	18	pile	17
cpile	17	rcol	18
define	20	right	16
delim	19	roman	12
dot	13	rpile	17
dotdot	13	size	12
down	21	sqrt	10
dyad	13	sub	7
fat	12	sup	7
font	12	tdefine	20
from	11	tilde	13
fwd	21	to	11
gfont	12	under	13
gsize	12	up	21
hat	13	vec	13
italic	12	~, ^	4, 6
lcol	18	{ }	8
left	16	"..."	8, 14
lineup	15		

24. Troubleshooting

If you make a mistake in an equation, like leaving out a brace (very common) or having one too many (very common) or having a *sup* with nothing before it (common), EQN will tell you with the message

syntax error between lines x and y, file z

where x and y are approximately the lines between which the trouble occurred, and z is the name of the file in question. The line numbers are approximate — look nearby as well. There are also self-explanatory messages that arise if you leave out a quote or try to run EQN on a non-existent file.

If you want to check a document before actually printing it (on UNIX only),

```
eqn files >/dev/null
```

will throw away the output but print the messages.

If you use something like dollar signs as delimiters, it is easy to leave one out. This causes very strange troubles. The program *checkeq* (on GCOS, use *./checkeq* instead) checks for misplaced or missing dollar signs and similar troubles.

In-line equations can only be so big because of an internal buffer in TROFF. If you get a message "word overflow", you have exceeded this limit. If you print the equation as a displayed equation this message will usually go away. The message "line overflow" indicates you have exceeded an even bigger buffer. The only cure for this is to break the equation into two separate ones.

On a related topic, EQN does not break equations by itself — you must split long equations up across multiple lines by yourself, marking each by a separate .EQ ... .EN sequence. EQN does warn about equations that are too long to fit on one line.

25. Use on UNIX

To print a document that contains mathematics on the UNIX typesetter,

```
eqn files | troff
```

If there are any TROFF options, they go after the TROFF part of the command. For example,

```
eqn files | troff -ms
```

To run the same document on the GCOS typesetter, use

```
eqn files | troff -g (other options) | gcat
```

A compatible version of EQN can be used on devices like teletypes and DASI and GSI terminals which have half-line forward and reverse capabilities. To print equations on a Model 37 teletype, for example, use

```
neqn files | nroff
```

The language for equations recognized by NEQN is identical to that of EQN, although of course the output is more restricted.

To use a GSI or DASI terminal as the output device,

```
neqn files | nroff -Tx
```

where *x* is the terminal type you are using, such as *300* or *300S*.

EQN and NEQN can be used with the TBL program[2] for setting tables that contain mathematics. Use TBL before [N]EQN, like this:

```
tbl files | eqn | troff
tbl files | neqn | nroff
```

26. Acknowledgments

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to extend TROFF to make our task easier, and for his continuous assistance during the development and evolution of EQN. We are also grateful to A. V. Aho for advice on language design, to S. C. Johnson for assistance with the YACC compiler-compiler, and to all the EQN users who have made helpful suggestions and criticisms.

References

[1] J. F. Ossanna, "NROFF/TROFF User's Manual", Bell Laboratories Computing Science Technical Report #54, 1976.

[2] M. E. Lesk, "Typing Documents on UNIX", Bell Laboratories, 1976.

[3] M. E. Lesk, "TBL — A Program for Setting Tables", Bell Laboratories Computing Science Technical Report #49, 1976.

Tbl — A Program to Format Tables

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Tbl is a document formatting preprocessor for *troff* or *nroff* which makes even fairly complex tables easy to specify and enter. It is available on the PDP-11 UNIX* system and on Honeywell 6000 GCOS. Tables are made up of columns which may be independently centered, right-adjusted, left-adjusted, or aligned by decimal points. Headings may be placed over single columns or groups of columns. A table entry may contain equations, or may consist of several rows of text. Horizontal or vertical lines may be drawn as desired in the table, and any table or element may be enclosed in a box. For example:

1970 Federal Budget Transfers (in billions of dollars)			
State	Taxes collected	Money spent	Net
New York	22.91	21.35	−1.56
New Jersey	8.33	6.96	−1.37
Connecticut	4.12	3.10	−1.02
Maine	0.74	0.67	−0.07
California	22.29	22.42	+0.13
New Mexico	0.70	1.49	+0.79
Georgia	3.30	4.28	+0.98
Mississippi	1.15	2.32	+1.17
Texas	9.33	11.13	+1.80

January 16, 1979

* UNIX is a Trademark/Service Mark of the Bell System

Tbl — A Program to Format Tables

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction.

Tbl turns a simple description of a table into a *troff* or *nroff* [1] program (list of commands) that prints the table. *Tbl* may be used on the PDP-11 UNIX [2] system and on the Honeywell 6000 GCOS system. It attempts to isolate a portion of a job that it can successfully handle and leave the remainder for other programs. Thus *tbl* may be used with the equation formatting program *eqn* [3] or various layout macro packages [4,5,6], but does not duplicate their functions.

This memorandum is divided into two parts. First we give the rules for preparing *tbl* input; then some examples are shown. The description of rules is precise but technical, and the beginning user may prefer to read the examples first, as they show some common table arrangements. A section explaining how to invoke *tbl* precedes the examples. To avoid repetition, henceforth read *troff* as "*troff* or *nroff.*"

The input to *tbl* is text for a document, with tables preceded by a ".TS" (table start) command and followed by a ".TE" (table end) command. *Tbl* processes the tables, generating *troff* formatting commands, and leaves the remainder of the text unchanged. The ".TS" and ".TE" lines are copied, too, so that *troff* page layout macros (such as the memo formatting macros [4]) can use these lines to delimit and place tables as they see fit. In particular, any arguments on the ".TS" or ".TE" lines are copied but otherwise ignored, and may be used by document layout macro commands.

The format of the input is as follows:

```
text
.TS
table
.TE
text
.TS
table
.TE
text
. . .
```

where the format of each table is as follows:

```
.TS
options ;
format .
data
.TE
```

Each table is independent, and must contain formatting information followed by the data to be entered in the table. The formatting information, which describes the individual columns and rows of the table, may be preceded by a few options that affect the entire table. A detailed description of tables is given in the next section.

Input commands.

As indicated above, a table contains, first, global options, then a format section describing the layout of the table entries, and then the data to be printed. The format and data are always required, but not the options. The various parts of the table are entered as follows:

1) OPTIONS. There may be a single line of options affecting the whole table. If present, this line must follow the .TS line immediately and must contain a list of option names separated by spaces, tabs, or commas, and must be terminated by a semicolon. The allowable options are:

center	— center the table (default is left-adjust);
expand	— make the table as wide as the current line length;
box	— enclose the table in a box;
allbox	— enclose each item in the table in a box;
doublebox	— enclose the table in two boxes;
tab (*x*)	— use *x* instead of tab to separate data items.
linesize (*n*)	— set lines or rules (e.g. from **box**) in *n* point type;
delim (*xy*)	— recognize *x* and *y* as the *eqn* delimiters.

The *tbl* program tries to keep boxed tables on one page by issuing appropriate "need" (.*ne*) commands. These requests are calculated from the number of lines in the tables, and if there are spacing commands embedded in the input, these requests may be inaccurate; use normal *troff* procedures, such as keep-release macros, in that case. The user who must have a multi-page boxed table should use macros designed for this purpose, as explained below under 'Usage.'

2) FORMAT. The format section of the table specifies the layout of the columns. Each line in this section corresponds to one line of the table (except that the last line corresponds to all following lines up to the next .T&, if any — see below), and each line contains a key-letter for each column of the table. It is good practice to separate the key letters for each column by spaces or tabs. Each key-letter is one of the following:

L or **l**	to indicate a left-adjusted column entry;
R or **r**	to indicate a right-adjusted column entry;
C or **c**	to indicate a centered column entry;
N or **n**	to indicate a numerical column entry, to be aligned with other numerical entries so that the units digits of numbers line up;
A or **a**	to indicate an alphabetic subcolumn; all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column (see example on page 12);
S or **s**	to indicate a spanned heading, i.e. to indicate that the entry from the previous column continues across this column (not allowed for the first column, obviously); or
^	to indicate a vertically spanned heading, i.e. to indicate that the entry from the previous row continues down through this row. (Not allowed for the first row of the table, obviously).

When numerical alignment is specified, a location for the decimal point is sought. The rightmost dot (.) adjacent to a digit is used as a decimal point; if there is no dot adjoining a digit, the rightmost digit is used as a units digit; if no alignment is indicated, the item is centered in the column. However, the special non-printing character string \& may be used to override unconditionally dots and digits, or to align alphabetic data; this string lines up where a dot normally would, and then disappears from the final output. In the example below, the items shown at the left will be aligned (in a numerical column) as shown on the right:

```
13              13
4.2              4.2
26.4.12        26.4.12
abc               abc
abc\&            abc
43\&3.22         433.22
749.12          749.12
```

Note: If numerical data are used in the same column with wider **L** or **r** type table entries, the widest *number* is centered relative to the wider **L** or **r** items (**L** is used instead of **l** for readability; they have the same meaning as key-letters). Alignment within the numerical items is preserved. This is similar to the behavior of **a** type data, as explained above. However, alphabetic subcolumns (requested by the **a** key-letter) are always slightly indented relative to **L** items; if necessary, the column width is increased to force this. This is not true for **n** type entries.

Warning: the **n** and **a** items should not be used in the same column.

For readability, the key-letters describing each column should be separated by spaces. The end of the format section is indicated by a period. The layout of the key-letters in the format section resembles the layout of the actual data in the table. Thus a simple format might appear as:

```
c s s
l n n .
```

which specifies a table of three columns. The first line of the table contains a heading centered across all three columns; each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data. A sample table in this format might be:

Overall title		
Item-a	34.22	9.1
Item-b	12.65	.02
Items: c,d,e	23	5.8
Total	69.87	14.92

There are some additional features of the key-letter system:

Horizontal lines — A key-letter may be replaced by '_' (underscore) to indicate a horizontal line in place of the corresponding column entry, or by '=' to indicate a double horizontal line. If an adjacent column contains a horizontal line, or if there are vertical lines adjoining this column, this horizontal line is extended to meet the nearby lines. If any data entry is provided for this column, it is ignored and a warning message is printed.

Vertical lines — A vertical bar may be placed between column key-letters. This will cause a vertical line between the corresponding columns of the table. A vertical bar to the left of the first key-letter or to the right of the last one produces a line at the edge of the table. If two vertical bars appear between key-letters, a double vertical line is drawn.

Space between columns — A number may follow the key-letter. This indicates the amount of separation between this column and the next column. The number normally specifies the separation in *ens* (one en is about the width of the letter 'n').* If the "expand" option is used, then these numbers are multiplied by a constant such that the table is as wide as the current line length. The default column separation

* More precisely, an en is a number of points (1 point = 1/72 inch) equal to half the current type size.

number is 3. If the separation is changed the worst case (largest space requested) governs.

Vertical spanning — Normally, vertically spanned items extending over several rows of the table are centered in their vertical range. If a key-letter is followed by **t** or **T**, any corresponding vertically spanned item will begin at the top line of its range.

Font changes — A key-letter may be followed by a string containing a font name or number preceded by the letter **f** or **F**. This indicates that the corresponding column should be in a different font from the default font (usually Roman). All font names are one or two letters; a one-letter font name should be separated from whatever follows by a space or tab. The single letters **B**, **b**, **I**, and **i** are shorter synonyms for **fB** and **fI**. Font change commands given with the table entries override these specifications.

Point size changes — A key-letter may be followed by the letter **p** or **P** and a number to indicate the point size of the corresponding table entries. The number may be a signed digit, in which case it is taken as an increment or decrement from the current point size. If both a point size and a column separation value are given, one or more blanks must separate them.

Vertical spacing changes — A key-letter may be followed by the letter **v** or **V** and a number to indicate the vertical line spacing to be used within a multi-line corresponding table entry. The number may be a signed digit, in which case it is taken as an increment or decrement from the current vertical spacing. A column separation value must be separated by blanks or some other specification from a vertical spacing request. This request has no effect unless the corresponding table entry is a text block (see below).

Column width indication — A key-letter may be followed by the letter **w** or **W** and a width value in parentheses. This width is used as a minimum column width. If the largest element in the column is not as wide as the width value given after the w, the largest element is assumed to be that wide. If the largest element in the column is wider than the specified value, its width is used. The width is also used as a default line length for included text blocks. Normal *troff* units can be used to scale the width value; if none are used, the default is ens. If the width specification is a unitless integer the parentheses may be omitted. If the width value is changed in a column, the *last* one given controls.

Equal width columns — A key-letter may be followed by the letter **e** or **E** to indicate equal width columns. All columns whose key-letters are followed by **e** or **E** are made the same width. This permits the user to get a group of regularly spaced columns.

Note: The order of the above features is immaterial; they need not be separated by spaces, except as indicated above to avoid ambiguities involving point size and font changes. Thus a numerical column entry in italic font and 12 point type with a minimum width of 2.5 inches and separated by 6 ens from the next column could be specified as

```
np12w(2.5i)fI 6
```

Alternative notation — Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats may be given on the same line, separated by commas, so that the format for the example above might have been written:

```
c s s, l n n .
```

Default — Column descriptors missing from the end of a format line are assumed to be L. The longest line in the format section, however, defines the number of columns in the table; extra columns in the data are ignored silently.

3) DATA. The data for the table are typed after the format. Normally, each table line is typed as one line of data. Very long input lines can be broken: any line whose last character is \ is combined with the following line (and the \ vanishes). The data for different columns (the table entries) are separated by tabs, or by whatever character has been specified in the option *tabs* option. There are a few special cases:

Troff commands within tables — An input line beginning with a '.' followed by anything but a number is assumed to be a command to *troff* and is passed through unchanged, retaining its position in the table. So, for example, space within a table may be produced by ".sp" commands in the data.

Full width horizontal lines — An input *line* containing only the character _ (underscore) or = (equal sign) is taken to be a single or double line, respectively, extending the full width of the *table.*

Single column horizontal lines — An input table *entry* containing only the character _ or = is taken to be a single or double line extending the full width of the *column.* Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain these characters explicitly in a column, either precede them by \& or follow them by a space before the usual tab or newline.

Short horizontal lines — An input table *entry* containing only the string _ is taken to be a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

Repeated characters — An input table *entry* containing only a string of the form \R*x* where *x* is any character is replaced by repetitions of the character *x* as wide as the data in the column. The sequence of *x*'s is not extended to meet adjoining columns.

Vertically spanned items — An input table entry containing only the character string \^ indicates that the table entry immediately above spans downward over this row. It is equivalent to a table format key-letter of '^'.

Text blocks — In order to include a block of text as a table entry, precede it by T{ and follow it by T}. Thus the sequence

```
. . . T{
block of
text
T} . . .
```

is the way to enter, as a single entry in the table, something that cannot conveniently be typed as a simple string between tabs. Note that the T} end delimiter must begin a line; additional columns of data may follow after a tab on the same line. See the example on page 10 for an illustration of included text blocks in a table. If more than twenty or thirty text blocks are used in a table, various limits in the *troff* program are likely to be exceeded, producing diagnostics such as 'too many string/macro names' or 'too many number registers.'

Text blocks are pulled out from the table, processed separately by *troff*, and replaced in the table as a solid block. If no line length is specified in the *block of text* itself, or in the table format, the default is to use $L \times C/(N+1)$ where L is the current line length, C is the number of table columns spanned by the text, and N is the total number of columns in the table. The other parameters (point size, font, etc.) used in setting the *block of text* are those in effect at the beginning of the table (including the effect of the ".TS" macro) and any table format specifications of size, spacing and font, using the p, v and f modifiers to the column key-letters. Commands within the text block itself are also recognized, of course. However, *troff* commands within the table data but not within the text block do not affect that block.

Warnings: — Although any number of lines may be present in a table, only the first 200 lines are used in calculating the widths of the various columns. A multi-page table, of course, may be arranged as several single-page tables if this proves to be a problem. Other difficulties with formatting may arise because, in the calculation of column widths all table entries are assumed to be in the font and size being used when the ".TS" command was encountered, except for font and size changes indicated (a) in the table format section and (b) within the table data (as in the entry \s+3\fIdata\fP\s0). Therefore, although arbitrary *troff* requests may be sprinkled in a table, care must be taken to avoid confusing the width calculations; use requests such as '.ps' with care.

4) ADDITIONAL COMMAND LINES. If the format of a table must be changed after many similar lines, as with sub-headings or summarizations, the ".T&" (table continue) command can be used to change column parameters. The outline of such a table input is:

```
.TS
options ;
format .
data
. . .
.T&
format .
data
.T&
format .
data
.TE
```

as in the examples on pages 10 and 12. Using this procedure, each table line can be close to its corresponding format line.

Warning: it is not possible to change the number of columns, the space between columns, the global options such as *box*, or the selection of columns to be made equal width.

Usage.

On UNIX, *tbl* can be run on a simple table with the command

```
tbl input-file | troff
```

but for more complicated use, where there are several input files, and they contain equations and *ms* memorandum layout commands as well as tables, the normal command would be

```
tbl file-1 file-2 . . . | eqn | troff −ms
```

and, of course, the usual options may be used on the *troff* and *eqn* commands. The usage for *nroff* is similar to that for *troff*, but only TELETYPE® Model 37 and Diablo-mechanism (DASI or GSI) terminals can print boxed tables directly.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special *−TX* command line option to *tbl* which produces output that does not have fractional line motions in it. The only other command line options recognized by *tbl* are *−ms* and *−mm* which are turned into commands to fetch the corresponding macro files; usually it is more convenient to place these arguments on the *troff* part of the command line, but they are accepted by *tbl* as well.

Note that when *eqn* and *tbl* are used together on the same file *tbl* should be used first. If there are no equations within tables, either order works, but it is usually faster to run *tbl* first, since *eqn* normally produces a larger expansion of the input than *tbl*. However, if there are equations within tables (using the *delim* mechanism in *eqn*), *tbl* must be first or the output will be scrambled. Users must also beware of using equations in **n**-style columns; this is nearly always wrong, since *tbl* attempts to split numerical format items into two parts and this is not possible with equations. The user can defend against this by giving the *delim(xx)* table option; this prevents splitting of numerical columns within the delimiters. For example, if the *eqn* delimiters are *$$*, giving *delim($$)* a numerical column such as "1245 $+- 16$" will be divided after 1245, not after 16.

Tbl limits tables to twenty columns; however, use of more than 16 numerical columns may fail because of limits in *troff*, producing the 'too many number registers' message. *Troff* number registers used by *tbl* must be avoided by the user within tables; these include two-digit names from 31 to 99, and names of the forms #*x*, *x*+, *x*|, ^*x*, and *x*−, where *x* is any lower case letter. The names ##, #−, and #^ are also used in certain circumstances. To conserve number register names, the **n** and **a** formats share a register; hence the restriction above that they may not be used in the same column.

For aid in writing layout macros, *tbl* defines a number register TW which is the table width; it is defined by the time that the ".TE" macro is invoked and may be used in the expansion of that macro. More importantly, to assist in laying out multi-page boxed tables the macro T# is defined to produce the bottom lines and side lines of a boxed table, and then invoked at its end. By use of this macro in the page footer a multi-page table can be boxed. In particular, the *ms* macros can be used to print a multi-page boxed table with a repeated heading by giving the argument H to the ".TS" macro. If the table start macro is written

```
     .TS H
```

a line of the form

```
     .TH
```

must be given in the table after any table heading (or at the start if none). Material up to the ".TH" is placed at the top of each page of table; the remaining lines in the table are placed on several pages as required. Note that this is *not* a feature of *tbl*, but of the *ms* layout macros.

Examples.

Here are some examples illustrating features of *tbl*. The symbol Ⓣ in the input represents a tab character.

Input:

```
.TS
box;
c c c
l l l.
LanguageⓉAuthorsⓉRuns on

FortranⓉManyⓉAlmost anything
PL/1ⓉIBMⓉ360/370
CⓉBTLⓉ11/45,H6000,370
BLISSⓉCarnegie-MellonⓉPDP-10,11
IDSⓉHoneywellⓉH6000
PascalⓉStanfordⓉ370
.TE
```

Output:

Language	Authors	Runs on
Fortran	Many	Almost anything
PL/1	IBM	360/370
C	BTL	11/45,H6000,370
BLISS	Carnegie-Mellon	PDP-10,11
IDS	Honeywell	H6000
Pascal	Stanford	370

Input:

```
.TS
allbox;
c s s
c c c
n n n.
AT&T Common Stock
YearⓣPriceⓣDividend
1971ⓣ41-54ⓣ$2.60
2ⓣ41-54ⓣ2.70
3ⓣ46-55ⓣ2.87
4ⓣ40-53ⓣ3.24
5ⓣ45-52ⓣ3.40
6ⓣ51-59ⓣ.95*
.TE
* (first quarter only)
```

Output:

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

Input:

```
.TS
box;
c s s
c|c|c
l|l|n.
Major New York Bridges
=
BridgeⓣDesignerⓣLength
_
BrooklynⓣJ. A. Roeblingⓣ1595
ManhattanⓣG. Lindenthalⓣ1470
WilliamsburgⓣL. L. Buckⓣ1600
_
QueensboroughⓣPalmer &ⓣ1182
ⓣ  Hornbostel
_
ⓣⓣ1380
TriboroughⓣO. H. Ammannⓣ_
ⓣⓣ383
_
Bronx WhitestoneⓣO. H. Ammannⓣ2300
Throgs NeckⓣO. H. Ammannⓣ1800
_
George WashingtonⓣO. H. Ammannⓣ3500
.TE
```

Output:

Major New York Bridges		
Bridge	Designer	Length
Brooklyn	J. A. Roebling	1595
Manhattan	G. Lindenthal	1470
Williamsburg	L. L. Buck	1600
Queensborough	Palmer &	1182
	Hornbostel	
		1380
Triborough	O. H. Ammann	
		383
Bronx Whitestone	O. H. Ammann	2300
Throgs Neck	O. H. Ammann	1800
George Washington	O. H. Ammann	3500

Input:

```
.TS
c c
np-2 | n | .
ⓉStack
Ⓣ_
1Ⓣ46
Ⓣ_
2Ⓣ23
Ⓣ_
3Ⓣ15
Ⓣ_
4Ⓣ6.5
Ⓣ_
5Ⓣ2.1
Ⓣ_
.TE
```

Output:

	Stack
1	46
2	23
3	15
4	6.5
5	2.1

Input:

```
.TS
box;
L L L
L L _
L L |LB
L L _
L L L.
januaryⓉfebruaryⓉmarch
aprilⓉmay
juneⓉjulyⓉMonths
augustⓉseptember
octoberⓉnovemberⓉdecember
.TE
```

Output:

january	february	march
april	may	
june	july	**Months**
august	september	
october	november	december

Input:

```
.TS
box;
cfB s s s.
Composition of Foods
_
.T&
c |c s s
c |c s s
c |c |c |c.
FoodⓉPercent by Weight
\^Ⓣ_
\^ⓉProteinⓉFatⓉCarbo-
\^Ⓣ\^Ⓣ\^Ⓣhydrate
_
.T&
l |n |n |n.
ApplesⓉ.4Ⓣ.5Ⓣ13.0
HalibutⓉ18.4Ⓣ5.2Ⓣ. . .
Lima beansⓉ7.5Ⓣ.8Ⓣ22.0
MilkⓉ3.3Ⓣ4.0Ⓣ5.0
MushroomsⓉ3.5Ⓣ.4Ⓣ6.0
Rye breadⓉ9.0Ⓣ.6Ⓣ52.7
.TE
```

Output:

Composition of Foods			
Food	Percent by Weight		
	Protein	Fat	Carbo-hydrate
Apples	.4	.5	13.0
Halibut	18.4	5.2	...
Lima beans	7.5	.8	22.0
Milk	3.3	4.0	5.0
Mushrooms	3.5	.4	6.0
Rye bread	9.0	.6	52.7

Input:

```
.TS
allbox;
cfI s  s
c   cw(1i)  cw(1i)
lp9 lp9 lp9.
New York Area Rocks
EraⓉFormationⓉAge (years)
PrecambrianⓉReading ProngⓉ>1 billion
PaleozoicⓉManhattan ProngⓉ400 million
MesozoicⓉT{
.na
Newark Basin, incl.
Stockton, Lockatong, and Brunswick
formations; also Watchungs
and Palisades.
T}Ⓣ200 million
CenozoicⓉCoastal PlainⓉT{
On Long Island 30,000 years;
Cretaceous sediments redeposited
by recent glaciation.
.ad
T}
.TE
```

Output:

New York Area Rocks		
Era	Formation	Age (years)
Precambrian	Reading Prong	>1 billion
Paleozoic	Manhattan Prong	400 million
Mesozoic	Newark Basin, incl. Stockton, Lockatong, and Brunswick formations; also Watchungs and Palisades.	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years; Cretaceous sediments redeposited by recent glaciation.

Input:

```
.EQ
delim $$
.EN

. . .

.TS
doublebox;
c c
l l.
NameⓉDefinition
.sp
.vs +2p
GammaⓉ$GAMMA (z) = int sub 0 sup inf  t sup {z-1} e sup -t dt$
SineⓉ$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
ErrorⓉ$ roman erf (z) = 2 over sqrt pi int sub 0 sup z e sup {-t sup 2} dt$
BesselⓉ$ J sub 0 (z) = 1 over pi int sub 0 sup pi cos ( z sin theta ) d theta $
ZetaⓉ$ zeta (s) = sum from k=1 to inf k sup -s ~( Re~s > 1)$
.vs -2p
.TE
```

Output:

Name	Definition
Gamma	$\Gamma(z)=\int_0^\infty t^{z-1}e^{-t}dt$
Sine	$\sin(x)=\frac{1}{2i}(e^{ix}-e^{-ix})$
Error	$\mathrm{erf}(z)=\frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$
Bessel	$J_0(z)=\frac{1}{\pi}\int_0^\pi \cos(z\sin\theta)d\theta$
Zeta	$\zeta(s)=\sum_{k=1}^\infty k^{-s}\ (\mathrm{Re}\ s>1)$

Input:

```
.TS
box, tab(:);
cb s s s s
cp-2 s s s s
c||c|c|c|c
c||c|c|c|c
r2||n2|n2|n2|n.
Readability of Text
Line Width and Leading for 10-Point Type
=
Line:Set:1-Point:2-Point:4-Point
Width:Solid:Leading:Leading:Leading
_
9 Pica:\-9.3:\-6.0:\-5.3:\-7.1
14 Pica:\-4.5:\-0.6:\-0.3:\-1.7
19 Pica:\-5.0:\-5.1: 0.0:\-2.0
31 Pica:\-3.7:\-3.8:\-2.4:\-3.6
43 Pica:\-9.1:\-9.0:\-5.9:\-8.8
.TE
```

Output:

Readability of Text Line Width and Leading for 10-Point Type				
Line Width	Set Solid	1-Point Leading	2-Point Leading	4-Point Leading
9 Pica	−9.3	−6.0	−5.3	−7.1
14 Pica	−4.5	−0.6	−0.3	−1.7
19 Pica	−5.0	−5.1	0.0	−2.0
31 Pica	−3.7	−3.8	−2.4	−3.6
43 Pica	−9.1	−9.0	−5.9	−8.8

Input:

```
.TS
c s
cip-2 s
l n
a n.
Some London Transport Statistics
(Year 1964)
Railway route milesⓉ244
TubeⓉ66
Sub-surfaceⓉ22
SurfaceⓉ156
.sp .5
.T&
l r
a r.
Passenger traffic \- railway
JourneysⓉ674 million
Average lengthⓉ4.55 miles
Passenger milesⓉ3,066 million
.T&
l r
a r.
Passenger traffic \- road
JourneysⓉ2,252 million
Average lengthⓉ2.26 miles
Passenger milesⓉ5,094 million
.T&
l n
a n.
.sp .5
VehiclesⓉ12,521
Railway motor carsⓉ2,905
Railway trailer carsⓉ1,269
Total railwayⓉ4,174
OmnibusesⓉ8,347
.T&
l n
a n.
.sp .5
StaffⓉ73,739
Administrative, etc.Ⓣ5,582
Civil engineeringⓉ5,134
Electrical eng.Ⓣ1,714
Mech. eng. \- railwayⓉ4,310
Mech. eng. \- roadⓉ9,152
Railway operationsⓉ8,930
Road operationsⓉ35,946
OtherⓉ2,971
.TE
```

Output:

```
        Some London Transport Statistics
                   (Year 1964)
Railway route miles                    244
   Tube                                 66
   Sub-surface                          22
   Surface                             156

Passenger traffic – railway
   Journeys                    674 million
   Average length               4.55 miles
   Passenger miles           3,066 million
Passenger traffic – road
   Journeys                  2,252 million
   Average length               2.26 miles
   Passenger miles           5,094 million

Vehicles                            12,521
   Railway motor cars                2,905
   Railway trailer cars              1,269
   Total railway                     4,174
   Omnibuses                         8,347

Staff                               73,739
   Administrative, etc.              5,582
   Civil engineering                 5,134
   Electrical eng.                   1,714
   Mech. eng. – railway              4,310
   Mech. eng. – road                 9,152
   Railway operations                8,930
   Road operations                  35,946
   Other                             2,971
```

Input:

```
.ps 8
.vs 10p
.TS
center box;
c s s
ci s s
c c c
lB l n.
New Jersey Representatives
(Democrats)
.sp .5
NameⓉOffice addressⓉPhone
.sp .5
James J. FlorioⓉ23 S. White Horse Pike, Somerdale 08083Ⓣ609-627-8222
William J. HughesⓉ2920 Atlantic Ave., Atlantic City 08401Ⓣ609-345-4844
James J. HowardⓉ801 Bangs Ave., Asbury Park 07712Ⓣ201-774-1600
Frank Thompson, Jr.Ⓣ10 Rutgers Pl., Trenton 08618Ⓣ609-599-1619
Andrew MaguireⓉ115 W. Passaic St., Rochelle Park 07662Ⓣ201-843-0240
Robert A. RoeⓉU.S.P.O., 194 Ward St., Paterson 07510Ⓣ201-523-5152
Henry HelstoskiⓉ666 Paterson Ave., East Rutherford 07073Ⓣ201-939-9090
Peter W. Rodino, Jr.ⓉSuite 1435A, 970 Broad St., Newark 07102Ⓣ201-645-3213
Joseph G. MinishⓉ308 Main St., Orange 07050Ⓣ201-645-6363
Helen S. MeynerⓉ32 Bridge St., Lambertville 08530Ⓣ609-397-1830
Dominick V. DanielsⓉ895 Bergen Ave., Jersey City 07306Ⓣ201-659-7700
Edward J. PattenⓉNatl. Bank Bldg., Perth Amboy 08861Ⓣ201-826-4610
.sp .5
.T&
ci s s
lB l n.
(Republicans)
.sp .5v
Millicent FenwickⓉ41 N. Bridge St., Somerville 08876Ⓣ201-722-8200
Edwin B. ForsytheⓉ301 Mill St., Moorestown 08057Ⓣ609-235-6622
Matthew J. RinaldoⓉ1961 Morris Ave., Union 07083Ⓣ201-687-4235
.TE
.ps 10
.vs 12p
```

Output:

New Jersey Representatives
(Democrats)

Name	Office address	Phone
James J. Florio	23 S. White Horse Pike, Somerdale 08083	609-627-8222
William J. Hughes	2920 Atlantic Ave., Atlantic City 08401	609-345-4844
James J. Howard	801 Bangs Ave., Asbury Park 07712	201-774-1600
Frank Thompson, Jr.	10 Rutgers Pl., Trenton 08618	609-599-1619
Andrew Maguire	115 W. Passaic St., Rochelle Park 07662	201-843-0240
Robert A. Roe	U.S.P.O., 194 Ward St., Paterson 07510	201-523-5152
Henry Helstoski	666 Paterson Ave., East Rutherford 07073	201-939-9090
Peter W. Rodino, Jr.	Suite 1435A, 970 Broad St., Newark 07102	201-645-3213
Joseph G. Minish	308 Main St., Orange 07050	201-645-6363
Helen S. Meyner	32 Bridge St., Lambertville 08530	609-397-1830
Dominick V. Daniels	895 Bergen Ave., Jersey City 07306	201-659-7700
Edward J. Patten	Natl. Bank Bldg., Perth Amboy 08861	201-826-4610
	(Republicans)	
Millicent Fenwick	41 N. Bridge St., Somerville 08876	201-722-8200
Edwin B. Forsythe	301 Mill St., Moorestown 08057	609-235-6622
Matthew J. Rinaldo	1961 Morris Ave., Union 07083	201-687-4235

This is a paragraph of normal text placed here only to indicate where the left and right margins are. In this way the reader can judge the appearance of centered tables or expanded tables, and observe how such tables are formatted.

Input:

```
.TS
expand;
c s s s
c c c c
l l n n.
Bell Labs Locations
NameⓉAddressⓉArea CodeⓉPhone
HolmdelⓉHolmdel, N. J. 07733Ⓣ201Ⓣ949-3000
Murray HillⓉMurray Hill, N. J. 07974Ⓣ201Ⓣ582-6377
WhippanyⓉWhippany, N. J. 07981Ⓣ201Ⓣ386-3000
Indian HillⓉNaperville, Illinois 60540Ⓣ312Ⓣ690-2000
.TE
```

Output:

Bell Labs Locations

Name	Address	Area Code	Phone
Holmdel	Holmdel, N. J. 07733	201	949-3000
Murray Hill	Murray Hill, N. J. 07974	201	582-6377
Whippany	Whippany, N. J. 07981	201	386-3000
Indian Hill	Naperville, Illinois 60540	312	690-2000

Input:

```
.TS
box;
cb  s  s  s
c|c|c  s
ltiw(1i) | ltw(2i) | lp8 | lw(1.6i)p8.
Some Interesting Places

N̄ameⓉDescriptionⓉPractical Information

T̄{
American Museum of Natural History
T}ⓉT{
The collections fill 11.5 acres (Michelin) or 25 acres (MTA)
of exhibition halls on four floors. There is a full-sized replica
of a blue whale and the world's largest star sapphire (stolen in 1964).
T}ⓉHoursⓉ10-5, ex. Sun 11-5, Wed. to 9
\^Ⓣ\^ⓉLocationⓉT{
Central Park West & 79th St.
T}
\^Ⓣ\^ⓉAdmissionⓉDonation: $1.00 asked
\^Ⓣ\^ⓉSubwayⓉAA to 81st St.
\^Ⓣ\^ⓉTelephoneⓉ212-873-4225

B̄ronx ZooⓉT{
About a mile long and .6 mile wide, this is the largest zoo in America.
A lion eats 18 pounds
of meat a day while a sea lion eats 15 pounds of fish.
T}ⓉHoursⓉT{
10-4:30 winter, to 5:00 summer
T}
\^Ⓣ\^ⓉLocationⓉT{
185th St. & Southern Blvd, the Bronx.
T}
\^Ⓣ\^ⓉAdmissionⓉ$1.00, but Tu,We,Th free
\^Ⓣ\^ⓉSubwayⓉ2, 5 to East Tremont Ave.
\^Ⓣ\^ⓉTelephoneⓉ212-933-1759

B̄rooklyn MuseumⓉT{
Five floors of galleries contain American and ancient art.
There are American period rooms and architectural ornaments saved
from wreckers, such as a classical figure from Pennsylvania Station.
T}ⓉHoursⓉWed-Sat, 10-5, Sun 12-5
\^Ⓣ\^ⓉLocationⓉT{
Eastern Parkway & Washington Ave., Brooklyn.
T}
\^Ⓣ\^ⓉAdmissionⓉFree
\^Ⓣ\^ⓉSubwayⓉ2,3 to Eastern Parkway.
\^Ⓣ\^ⓉTelephoneⓉ212-638-5000

T̄{
New-York Historical Society
T}ⓉT{
All the original paintings for Audubon's
.I
Birds of America
.R
are here, as are exhibits of American decorative arts, New York history,
Hudson River school paintings, carriages, and glass paperweights.
T}ⓉHoursⓉT{
Tues-Fri & Sun, 1-5; Sat 10-5
T}
\^Ⓣ\^ⓉLocationⓉT{
Central Park West & 77th St.
T}
\^Ⓣ\^ⓉAdmissionⓉFree
\^Ⓣ\^ⓉSubwayⓉAA to 81st St.
\^Ⓣ\^ⓉTelephoneⓉ212-873-3400
.TE
```

Output:

Some Interesting Places			
Name	Description	Practical Information	
American Museum of Natural History	The collections fill 11.5 acres (Michelin) or 25 acres (MTA) of exhibition halls on four floors. There is a full-sized replica of a blue whale and the world's largest star sapphire (stolen in 1964).	Hours	10-5, ex. Sun 11-5, Wed. to 9
		Location	Central Park West & 79th St.
		Admission	Donation: $1.00 asked
		Subway	AA to 81st St.
		Telephone	212-873-4225
Bronx Zoo	About a mile long and .6 mile wide, this is the largest zoo in America. A lion eats 18 pounds of meat a day while a sea lion eats 15 pounds of fish.	Hours	10-4:30 winter, to 5:00 summer
		Location	185th St. & Southern Blvd, the Bronx.
		Admission	$1.00, but Tu,We,Th free
		Subway	2, 5 to East Tremont Ave.
		Telephone	212-933-1759
Brooklyn Museum	Five floors of galleries contain American and ancient art. There are American period rooms and architectural ornaments saved from wreckers, such as a classical figure from Pennsylvania Station.	Hours	Wed-Sat, 10-5, Sun 12-5
		Location	Eastern Parkway & Washington Ave., Brooklyn.
		Admission	Free
		Subway	2,3 to Eastern Parkway.
		Telephone	212-638-5000
New-York Historical Society	All the original paintings for Audubon's *Birds of America* are here, as are exhibits of American decorative arts, New York history, Hudson River school paintings, carriages, and glass paperweights.	Hours	Tues-Fri & Sun, 1-5; Sat 10-5
		Location	Central Park West & 77th St.
		Admission	Free
		Subway	AA to 81st St.
		Telephone	212-873-3400

Acknowledgments.

Many thanks are due to J. C. Blinn, who has done a large amount of testing and assisted with the design of the program. He has also written many of the more intelligible sentences in this document and helped edit all of it. All phototypesetting programs on UNIX are dependent on the work of the late J. F. Ossanna, whose assistance with this program in particular had been most helpful. This program is patterned on a table formatter originally written by J. F. Gimpel. The assistance of T. A. Dolotta, B. W. Kernighan, and J. N. Sturman is gratefully acknowledged.

References.

[1] J. F. Ossanna, *NROFF/TROFF User's Manual*, Computing Science Technical Report No. 54, Bell Laboratories, 1976.

[2] K. Thompson and D. M. Ritchie, "The UNIX Time-Sharing System," *Comm. ACM.* 17, pp. 365–75 (1974).

[3] B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. ACM.* 18, pp. 151–57 (1975).

[4] M. E. Lesk, *Typing Documents on UNIX*, UNIX Programmer's Manual, Volume 2.

[5] M. E. Lesk and B. W. Kernighan, *Computer Typesetting of Technical Journals on UNIX*, *Proc. AFIPS NCC*, vol. 46, pp. 879-888 (1977).

[6] J. R. Mashey and D. W. Smith, "Documentation Tools and Techniques," *Proc. 2nd Int. Conf. on Software Engineering*, pp. 177-181 (October, 1976).

List of Tbl Command Characters and Words

Command	*Meaning*	*Section*
a A	Alphabetic subcolumn	2
allbox	Draw box around all items	1
b B	Boldface item	2
box	Draw box around table	1
c C	Centered column	2
center	Center table in page	1
doublebox	Doubled box around table	1
e E	Equal width columns	2
expand	Make table full line width	1
f F	Font change	2
i I	Italic item	2
l L	Left adjusted column	2
n N	Numerical column	2
nnn	Column separation	2
p P	Point size change	2
r R	Right adjusted column	2
s S	Spanned item	2
t T	Vertical spanning at top	2
tab (*x*)	Change data separator character	1
T{ T}	Text block	3
v V	Vertical spacing change	2
w W	Minimum width value	2
.*xx*	Included *troff* command	3
|	Vertical line	2
||	Double vertical line	2
^	Vertical span	2
\\^	Vertical span	3
=	Double horizontal line	2,3
_	Horizontal line	2,3
_	Short horizontal line	3
\\R*x*	Repeat character	3

Some Applications of Inverted Indexes on the UNIX System

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction.

The UNIX† system has many utilities (e.g. *grep, awk, lex, egrep, fgrep,* ...) to search through files of text, but most of them are based on a linear scan through the entire file, using some deterministic automaton. This memorandum discusses a program which uses inverted indexes[1] and can thus be used on much larger data bases.

As with any indexing system, of course, there are some disadvantages; once an index is made, the files that have been indexed can not be changed without remaking the index. Thus applications are restricted to those making many searches of relatively stable data. Furthermore, these programs depend on hashing, and can only search for exact matches of whole keywords. It is not possible to look for arithmetic or logical expressions (e.g. "date greater than 1970") or for regular expression searching such as that in *lex.*[2]

Currently there are two uses of this software, the *refer* preprocessor to format references, and the *lookall* command to search through all text files on the UNIX system.

The remaining sections of this memorandum discuss the searching programs and their uses. Section 2 explains the operation of the searching algorithm and describes the data collected for use with the *lookall* command. The more important application, *refer* has a user's description in section 3. Section 4 goes into more detail on reference files for the benefit of those who wish to add references to data bases or write new *troff* macros for use with *refer.* The options to make *refer* collect identical citations, or otherwise relocate and adjust references, are described in section 5. The UNIX manual sections for *refer, lookall,* and associated commands are attached as appendices.

2. Searching.

The indexing and searching process is divided into two phases, each made of two parts. These are shown below.

A. Construct the index.

   (1) Find keys — turn the input files into a sequence of tags and keys, where each tag identifies a distinct item in the input and the keys for each such item are the strings under which it is to be indexed.

   (2) Hash and sort — prepare a set of inverted indexes from which, given a set of keys, the appropriate item tags can be found quickly.

B. Retrieve an item in response to a query.

---

†UNIX is a Trademark of Bell Laboratories.

1. D. Knuth, *The Art of Computer Programming: Vol. 3, Sorting and Searching,* Addison-Wesley, Reading, Mass. (1977). See section 6.5.

2. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (D).

(3) Search — Given some keys, look through the files prepared by the hashing and sorting facility and derive the appropriate tags.

(4) Deliver — Given the tags, find the original items. This completes the searching process.

The first phase, making the index, is presumably done relatively infrequently. It should, of course, be done whenever the data being indexed change. In contrast, the second phase, retrieving items, is presumably done often, and must be rapid.

An effort is made to separate code which depends on the data being handled from code which depends on the searching procedure. The search algorithm is involved only in steps (2) and (3), while knowledge of the actual data files is needed only by steps (1) and (4). Thus it is easy to adapt to different data files or different search algorithms.

To start with, it is necessary to have some way of selecting or generating keys from input files. For dealing with files that are basically English, we have a key-making program which automatically selects words and passes them to the hashing and sorting program (step 2). The format used has one line for each input item, arranged as follows:

```
name:start,length (tab) key1 key2 key3 ...
```

where *name* is the file name, *start* is the starting byte number, and *length* is the number of bytes in the entry.

These lines are the only input used to make the index. The first field (the file name, byte position, and byte count) is the tag of the item and can be used to retrieve it quickly. Normally, an item is either a whole file or a section of a file delimited by blank lines. After the tab, the second field contains the keys. The keys, if selected by the automatic program, are any alphanumeric strings which are not among the 100 most frequent words in English and which are not entirely numeric (except for four-digit numbers beginning 19, which are accepted as dates). Keys are truncated to six characters and converted to lower case. Some selection is needed if the original items are ver lrge. We normally just take the first *n* keys, with *n* less than 100 or so; this replaces any attempt at intelligent selection. One file in our system is a complete English dictionary; it would presumably be retrieved for all queries.

To generate an inverted index to the list of record tags and keys, the keys are hashed and sorted to produce an index. What is wanted, ideally, is a series of lists showing the tags associated with each key. To condense this, what is actually produced is a list showing the tags associated with each hash code, and thus with some set of keys. To speed up access and further save space, a set of three or possibly four files is produced. These files are:

File	Contents
entry	Pointers to posting file for each hash code
posting	Lists of tag pointers for each hash code
tag	Tags for each item
key	Keys for each item (optional)

The posting file comprises the real data: it contains a sequence of lists of items posted under each hash code. To speed up searching, the entry file is an array of pointers into the posting file, one per potential hash code. Furthermore, the items in the lists in the posting file are not referred to by their complete tag, but just by an address in the tag file, which gives the complete tags. The key file is optional and contains a copy of the keys used in the indexing.

The searching process starts with a query, containing several keys. The goal is to obtain all items which were indexed under these keys. The query keys are hashed, and the pointers in the entry file used to access the lists in the posting file. These lists are addresses in the tag file of documents posted under the hash codes derived from the query. The common items from all lists are determined; this must include the items indexed by every key, but may also contain some items which are false drops, since items referenced by the correct hash codes need not actually have contained the correct keys. Normally, if there are several keys in the query, there are not likely to be many false drops in the final combined list even though each hash code is somewhat ambiguous. The actual tags are then obtained from the tag file, and to guard against the possibility that an item has false-dropped on some hash code in the query, the original items are normally obtained from the delivery program (4) and the query keys checked against them by string comparison.

Usually, therefore, the check for bad drops is made against the original file. However, if the key derivation procedure is complex, it may be preferable to check against the keys fed to program (2). In this case the optional key file which contains the keys associated with each item is generated, and the item tag is supplemented by a string

;start,length

which indicates the starting byte number in the key file and the length of the string of keys for each item. This file is not usually necessary with the present key-selection program, since the keys always appear in the original document.

There is also an option (-C*n*) for coordination level searching. This retrieves items which match all but *n* of the query keys. The items are retrieved in the order of the number of keys that they match. Of course, *n* must be less than the number of query keys (nothing is retrieved unless it matches at least one key).

As an example, consider one set of 4377 references, comprising 660,000 bytes. This included 51,000 keys, of which 5,900 were distinct keys. The hash table is kept full to save space (at the expense of time); 995 of 997 possible hash codes were used. The total set of index files (no key file) included 171,000 bytes, about 26% of the original file size. It took 8 minutes of processor time to hash, sort, and write the index. To search for a single query with the resulting index took 1.9 seconds of processor time, while to find the same paper with a sequential linear search using *grep* (reading all of the tags and keys) took 12.3 seconds of processor time.

We have also used this software to index all of the English stored on our UNIX system. This is the index searched by the *lookall* command. On a typical day there were 29,000 files in our user file system, containing about 152,000,000 bytes. Of these 5,300 files, containing 32,000,000 bytes (about 21%) were English text. The total number of 'words' (determined mechanically) was 5,100,000. Of these 227,000 were selected as keys; 19,000 were distinct, hashing to 4,900 (of 5,000 possible) different hash codes. The resulting inverted file indexes used 845,000 bytes, or about 2.6% of the size of the original files. The particularly small indexes are caused by the fact that keys are taken from only the first 50 non-common words of some very long input files.

Even this large *lookall* index can be searched quickly. For example, to find this document by looking for the keys "lesk inverted indexes" required 1.7 seconds of processor time and system time. By comparison, just to search the 800,000 byte dictionary (smaller than even the inverted indexes, let alone the 32,000,000 bytes of text files) with *grep* takes 29 seconds of processor time. The *lookall* program is thus useful when looking for a document which you believe is stored on-line, but do not know where. For example, many memos from the Computing Science Research Center are in its UNIX file system, but it is often difficult to guess where a particular memo might be (it might have several authors, each with many directories, and have been worked on by a secretary with yet more directories). Instructions for the use of the *lookall* command are given in the manual section, shown in the appendix to this memorandum.

The only indexes maintained routinely are those of publication lists and all English files. To make other indexes, the programs for making keys, sorting them, searching the indexes, and delivering answers must be used. Since they are usually invoked as parts of higher-level commands, they are not in the default command directory, but are available to any user in the directory */usr/lib/refer*. Three programs are of interest: *mkey*, which isolates keys from input files; *inv*, which makes an index from a set of keys; and *hunt*, which searches the index and delivers the items. Note that the two parts of the retrieval phase are combined into one program, to avoid the excessive system work and delay which would result from running these as separate processes.

These three commands have a large number of options to adapt to different kinds of input. The user not interested in the detailed description that now follows may skip to section 3, which describes the *refer* program, a packaged-up version of these tools specifically oriented towards formatting references.

Make Keys. The program *mkey* is the key-making program corresponding to step (1) in phase A. Normally, it reads its input from the file names given as arguments, and if there are no arguments it reads from the standard input. It assumes that blank lines in the input delimit separate items, for each of which a different line of keys should be generated. The lines of keys are written on the standard output. Keys are any alphanumeric string in the input not among the most frequent words in English and not entirely numeric (except that all-numeric strings are acceptable if they are between 1900 and 1999). In the output, keys are translated to lower case, and truncated to six characters in length; any associated punctuation is removed. The following flag arguments are recognized by *mkey:*

−c *name*	Name of file of common words; default is */usr/lib/eign*.
−f *name*	Read a list of files from *name* and take each as an input argument.
−i *chars*	Ignore all lines which begin with '%' followed by any character in *chars*.
−k*n*	Use at most *n* keys per input item.
−l*n*	Ignore items shorter than *n* letters long.
−n*m*	Ignore as a key any word in the first *m* words of the list of common English words. The default is 100.
−s	Remove the labels *(file:start,length)* from the output; just give the keys. Used when searching rather than indexing.
−w	Each whole file is a separate item; blank lines in files are irrelevant.

The normal arguments for indexing references are the defaults, which are *−c /usr/lib/eign*, *−n100*, and *−l3*. For searching, the *−s* option is also needed. When the big *lookall* index of all English files is run, the options are *−w*, *−k50*, and *−f (filelist)*. When running on textual input, the *mkey* program processes about 1000 English words per processor second. Unless the *−k* option is used (and the input files are long enough for it to take effect) the output of *mkey* is comparable in size to its input.

Hash and Invert. The *inv* program computes the hash codes and writes the inverted files. It reads the output of *mkey* and writes the set of files described earlier in this section. It expects one argument, which is used as the base name for the three (or four) files to be written. Assuming an argument of *Index* (the default) the entry file is named *Index.ia*, the posting file *Index.ib*, the tag file *Index.ic*, and the key file (if present) *Index.id*. The *inv* program recognizes the following options:

−a	Append the new keys to a previous set of inverted files, making new files if there is no old set using the same base name.
−d	Write the optional key file. This is needed when you can not check for false drops by looking for the keys in the original inputs, i.e. when the key derivation procedure is complicated and the output keys are not words from the input files.
−h*n*	The hash table size is *n* (default 997); *n* should be prime. Making *n* bigger saves search time and spends disk space.

−i[u] *name*	Take input from file *name*, instead of the standard input; if **u** is present *name* is unlinked when the sort is started. Using this option permits the sort scratch space to overlap the disk space used for input keys.
−n	Make a completely new set of inverted files, ignoring previous files.
−p	Pipe into the sort program, rather than writing a temporary input file. This saves disk space and spends processor time.
−v	Verbose mode; print a summary of the number of keys which finished indexing.

About half the time used in *inv* is in the contained sort. Assuming the sort is roughly linear, however, a guess at the total timing for *inv* is 250 keys per second. The space used is usually of more importance: the entry file uses four bytes per possible hash (note the **−h** option), and the tag file around 15-20 bytes per item indexed. Roughly, the posting file contains one item for each key instance and one item for each possible hash code; the items are two bytes long if the tag file is less than 65336 bytes long, and the items are four bytes wide if the tag file is greater than 65536 bytes long. To minimize storage, the hash tables should be over-full; for most of the files indexed in this way, there is no other real choice, since the *entry* file must fit in memory.

Searching and Retrieving. The *hunt* program retrieves items from an index. It combines, as mentioned above, the two parts of phase (B): search and delivery. The reason why it is efficient to combine delivery and search is partly to avoid starting unnecessary processes, and partly because the delivery operation must be a part of the search operation in any case. Because of the hashing, the search part takes place in two stages: first items are retrieved which have the right hash codes associated with them, and then the actual items are inspected to determine false drops, i.e. to determine if anything with the right hash codes doesn't really have the right keys. Since the original item is retrieved to check on false drops, it is efficient to present it immediately, rather than only giving the tag as output and later retrieving the item again. If there were a separate key file, this argument would not apply, but separate key files are not common.

Input to *hunt* is taken from the standard input, one query per line. Each query should be in *mkey −s* output format; all lower case, no punctuation. The *hunt* program takes one argument which specifies the base name of the index files to be searched. Only one set of index files can be searched at a time, although many text files may be indexed as a group, of course. If one of the text files has been changed since the index, that file is searched with *fgrep;* this may occasionally slow down the searching, and care should be taken to avoid having many out of date files. The following option arguments are recognized by *hunt:*

−a	Give all output; ignore checking for false drops.
−C*n*	Coordination level *n;* retrieve items with not more than *n* terms of the input missing; default *C0*, implying that each search term must be in the output items.
−F[yn*d***]**	"−Fy" gives the text of all the items found; "−Fn" suppresses them. "−F*d*" where *d* is an integer gives the text of the first *d* items. The default is *−Fy.*
−g	Do not use *fgrep* to search files changed since the index was made; print an error comment instead.
−i *string*	Take *string* as input, instead of reading the standard input.
−l *n*	The maximum length of internal lists of candidate items is *n;* default 1000.
−o *string*	Put text output ("−Fy") in *string;* of use *only* when invoked from another program.

−p	Print hash code frequencies; mostly for use in optimizing hash table sizes.
−T[yn*d*]	"−Ty" gives the tags of the items found; "−Tn" suppresses them. "−T*d*" where *d* is an integer gives the first *d* tags. The default is −*Tn*.
−t *string*	Put tag output ("−Ty") in *string;* of use *only* when invoked from another program.

The timing of *hunt* is complex. Normally the hash table is overfull, so that there will be many false drops on any single term; but a multi-term query will have few false drops on all terms. Thus if a query is underspecified (one search term) many potential items will be examined and discarded as false drops, wasting time. If the query is overspecified (a dozen search terms) many keys will be examined only to verify that the single item under consideration has that key posted. The variation of search time with number of keys is shown in the table below. Queries of varying length were constructed to retrieve a particular document from the file of references. In the sequence to the left, search terms were chosen so as to select the desired paper as quickly as possible. In the sequence on the right, terms were chosen inefficiently, so that the query did not uniquely select the desired document until four keys had been used. The same document was the target in each case, and the final set of eight keys are also identical; the differences at five, six and seven keys are produced by measurement error, not by the slightly different key lists.

Efficient Keys				Inefficient Keys			
No. keys	Total drops (incl. false)	Retrieved Documents	Search time (seconds)	No. keys	Total drops (incl. false)	Retrieved Documents	Search time (seconds)
1	15	3	1.27	1	68	55	5.96
2	1	1	0.11	2	29	29	2.72
3	1	1	0.14	3	8	8	0.95
4	1	1	0.17	4	1	1	0.18
5	1	1	0.19	5	1	1	0.21
6	1	1	0.23	6	1	1	0.22
7	1	1	0.27	7	1	1	0.26
8	1	1	0.29	8	1	1	0.29

As would be expected, the optimal search is achieved when the query just specifies the answer; however, overspecification is quite cheap. Roughly, the time required by *hunt* can be approximated as 30 milliseconds per search key plus 75 milliseconds per dropped document (whether it is a false drop or a real answer). In general, overspecification can be recommended; it protects the user against additions to the data base which turn previously uniquely-answered queries into ambiguous queries.

The careful reader will have noted an enormous discrepancy between these times and the earlier quoted time of around 1.9 seconds for a search. The times here are purely for the search and retrieval: they are measured by running many searches through a single invocation of the *hunt* program alone. Usually, the UNIX command processor (the shell) must start both the *mkey* and *hunt* processes for each query, and arrange for the output of *mkey* to be fed to the *hunt* program. This adds a fixed overhead of about 1.7 seconds of processor time to any single search. Furthermore, remember that all these times are processor times: on a typical morning on our PDP 11/70 system, with about one dozen people logged on, to obtain 1 second of processor time for the search program took between 2 and 12 seconds of real time, with a median of 3.9 seconds and a mean of 4.8 seconds. Thus, although the work involved in a single search may be only 200 milliseconds, after you add the 1.7 seconds of startup processor time and then assume a 4:1 elapsed/processor time ratio, it will be 8 seconds before any response is printed.

3. Selecting and Formatting References for TROFF

The major application of the retrieval software is *refer,* which is a *troff* preprocessor like *eqn*.[3] It scans its input looking for items of the form

```
.[
imprecise citation
.]
```

where an imprecise citation is merely a string of words found in the relevant bibliographic citation. This is translated into a properly formatted reference. If the imprecise citation does not correctly identify a single paper (either selecting no papers or too many) a message is given. The data base of citations searched may be tailored to each system, and individual users may specify their own citation files. On our system, the default data base is accumulated from the publication lists of the members of our organization, plus about half a dozen personal bibliographies that were collected. The present total is about 4300 citations, but this increases steadily. Even now, the data base covers a large fraction of local citations.

For example, the reference for the *eqn* paper above was specified as

```
...
preprocessor like
.I eqn.
.[
kernighan cherry acm 1975
.]
It scans its input looking for items
...
```

This paper was itself printed using *refer*. The above input text was processed by *refer* as well as *tbl* and *troff* by the command

refer memo-file | tbl | troff −ms

and the reference was automatically translated into a correct citation to the ACM paper on mathematical typesetting.

The procedure to use to place a reference in a paper using *refer* is as follows. First, use the *lookbib* command to check that the paper is in the data base and to find out what keys are necessary to retrieve it. This is done by typing *lookbib* and then typing some potential queries until a suitable query is found. For example, had one started to find the *eqn* paper shown above by presenting the query

```
$ lookbib
kernighan cherry
(EOT)
```

lookbib would have found several items; experimentation would quickly have shown that the query given above is adequate. Overspecifying the query is of course harmless; it is even desirable, since it decreases the risk that a document added to the publication data base in the future will be retrieved in addition to the intended document. The extra time taken by even a grossly overspecified query is quite small. A particularly careful reader may have noticed that "acm" does not appear in the printed citation; we have supplemented some of the data base items with extra keywords, such as common abbreviations for journals or other sources, to aid in searching.

If the reference is in the data base, the query that retrieved it can be inserted in the text, between .[and .] brackets. If it is not in the data base, it can be typed into a private file of

3. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.* **18**, pp.151-157 (March 1975).

references, using the format discssed in the next section, and then the **−p** option used to search this private file. Such a command might read (if the private references are called *myfile*)

> *refer −p myfile document* | *tbl* | *eqn* | *troff −ms* . . .

where *tbl* and/or *eqn* could be omitted if not needed. The use of the *−ms* macros[4] or some other macro package, however, is essential. *Refer* only generates the data for the references; exact formatting is done by some macro package, and if none is supplied the references will not be printed.

By default, the references are numbered sequentially, and the *−ms* macros format references as footnotes at the bottom of the page. This memorandum is an example of that style. Other possibilities are discussed in section 5 below.

4. Reference Files.

A reference file is a set of bibliographic references usable with *refer*. It can be indexed using the software described in section 2 for fast searching. What *refer* does is to read the input document stream, looking for imprecise citation references. It then searches through reference files to find the full citations, and inserts them into the document. The format of the full citation is arranged to make it convenient for a macro package, such as the *−ms* macros, to format the reference for printing. Since the format of the final reference is determined by the desired style of output, which is determined by the macros used, *refer* avoids forcing any kind of reference appearance. All it does is define a set of string registers which contain the basic information about the reference; and provide a macro call which is expanded by the macro package to format the reference. It is the responsibility of the final macro package to see that the reference is actually printed; if no macros are used, and the output of *refer* fed untranslated to *troff*, nothing at all will be printed.

The strings defined by *refer* are taken directly from the files of references, which are in the following format. The references should be separated by blank lines. Each reference is a sequence of lines beginning with % and followed by a key-letter. The remainder of that line, and successive lines until the next line beginning with %, contain the information specified by the key-letter. In general, *refer* does not interpret the information, but merely presents it to the macro package for final formatting. A user with a separate macro package, for example, can add new key-letters or use the existing ones for other purposes without bothering *refer*.

The meaning of the key-letters given below, in particular, is that assigned by the *−ms* macros. Not all information, obviously, is used with each citation. For example, if a document is both an internal memorandum and a journal article, the macros ignore the memorandum version and cite only the journal article. Some kinds of information are not used at all in printing the reference; if a user does not like finding references by specifying title or author keywords, and prefers to add specific keywords to the citation, a field is available which is searched but not printed (K).

The key letters currently recognized by *refer* and *−ms*, with the kind of information implied, are:

4. M. E. Lesk, *Typing Documents on UNIX and GCOS: The -ms Macros for Troff*, Bell Laboratories internal memorandum (1977).

Key	Information specified	Key	Information specified
A	Author's name	N	Issue number
B	Title of book containing item	O	Other information
C	City of publication	P	Page(s) of article
D	Date	R	Technical report reference
E	Editor of book containing item	T	Title
G	Government (NTIS) ordering number	V	Volume number
I	Issuer (publisher)		
J	Journal name		
K	Keys (for searching)	X	or
L	Label	Y	or
M	Memorandum label	Z	Information not used by *refer*

For example, a sample reference could be typed as:

```
%T Bounds on the Complexity of the Maximal
Common Subsequence Problem
%Z ctr127
%A A. V. Aho
%A D. S. Hirschberg
%A J. D. Ullman
%J J. ACM
%V 23
%N 1
%P 1-12
%M abcd-78
%D Jan. 1976
```

Order is irrelevant, except that authors are shown in the order given. The output of *refer* is a stream of string definitions, one for each of the fields of each reference, as shown below.

```
.]-
.ds [A authors' names ...
.ds [T title ...
.ds [J journal ...
...
.] [ type-number
```

The *refer* program, in general, does not concern itself with the significance of the strings. The different fields are treated identically by *refer*, except that the X, Y and Z fields are ignored (see the −I option of *mkey*) in indexing and searching. All *refer* does is select the appropriate citation, based on the keys. The macro package must arrange the strings so as to produce an appropriately formatted citation. In this process, it uses the convention that the 'T' field is the title, the 'J' field the journal, and so forth.

The *refer* program does arrange the citation to simplify the macro package's job, however. The special macro **.]−** precedes the string definitions and the special macro **.][** follows. These are changed from the input **.[** and **.]** so that running the same file through *refer* again is harmless. The **.]−** macro can be used by the macro package to initialize. The **.][** macro, which should be used to print the reference, is given an argument *type-number* to indicate the kind of reference, as follows:

Value	Kind of reference
1	Journal article
2	Book
3	Article within book
4	Technical report
5	Bell Labs technical memorandum
0	Other

The type is determined by the presence or absence of particular fields in the citation (a journal article must have a 'J' field, a book must have an 'I' field, and so forth). To a small extent, this violates the above rule that *refer* does not concern itself with the contents of the citation; however, the classification of the citation in *troff* macros would require a relatively expensive and obscure program. Any macro writer may, of course, preserve consistency by ignoring the argument to the .][macro.

The reference is flagged in the text with the sequence

```
\*([.number\*(.]
```

where *number* is the footnote number. The strings [. and .] should be used by the macro package to format the reference flag in the text. These strings can be replaced for a particular footnote, as described in section 5. The footnote number (or other signal) is available to the reference macro .][as the string register [F. To simplify dealing with a text reference that occurs at the end of a sentence, *refer* treats a reference which follows a period in a special way. The period is removed, and the reference is preceded by a call for the string <. and followed by a call for the string >. For example, if a reference follows "end." it will appear as

```
end\*(<.\*([.number\*(.]\*(>.
```

where *number* is the footnote number. The macro package should turn either the string >. or <. into a period and delete the other one. This permits the output to have either the form "end[31]." or "end.^{31}" as the macro package wishes. Note that in one case the period precedes the number and in the other it follows the number.

In some cases users wish to suspend the searching, and merely use the reference macro formatting. That is, the user doesn't want to provide a search key between .[and .] brackets, but merely the reference lines for the appropriate document. Alternatively, the user can wish to add a few fields to those in the reference as in the standard file, or override some fields. Altering or replacing fields, or supplying whole references, is easily done by inserting lines beginning with %; any such line is taken as direct input to the reference processor rather than keys to be searched. Thus

```
.[
key1 key2 key3 ...
%Q New format item
%R Override report name
.]
```

makes the indicates changes to the result of searching for the keys. All of the search keys must be given before the first % line.

If no search keys are provided, an entire citation can be provided in-line in the text. For example, if the *eqn* paper citation were to be inserted in this way, rather than by searching for it in the data base, the input would read

```
...
preprocessor like
.I eqn.
.[
%A B. W. Kernighan
%A L. L. Cherry
%T A System for Typesetting Mathematics
%J Comm. ACM
%V 18
%N 3
%P 151-157
%D March 1975
.]
It scans its input looking for items
...
```

This would produce a citation of the same appearance as that resulting from the file search.

As shown, fields are normally turned into *troff* strings. Sometimes users would rather have them defined as macros, so that other *troff* commands can be placed into the data. When this is necessary, simply double the control character % in the data. Thus the input

```
.[
%V 23
%%M
Bell Laboratories,
Murray Hill, N.J. 07974
.]
```

is processed by *refer* into

```
.ds [V 23
.de [M
Bell Laboratories,
Murray Hill, N.J. 07974
..
```

The information after %%M is defined as a macro to be invoked by .[M while the information after %V is turned into a string to be invoked by *([V. At present *−ms* expects all information as strings.

5. Collecting References and other Refer Options

Normally, the combination of *refer* and *−ms* formats output as *troff* footnotes which are consecutively numbered and placed at the bottom of the page. However, options exist to place the references at the end; to arrange references alphabetically by senior author; and to indicate references by strings in the text of the form [Name1975a] rather than by number. Whenever references are not placed at the bottom of a page identical references are coalesced.

For example, the −e option to *refer* specifies that references are to be collected; in this case they are output whenever the sequence

```
.[
$LIST$
.]
```

is encountered. Thus, to place references at the end of a paper, the user would run *refer* with the *−e* option and place the above $LIST$ commands after the last line of the text. *Refer* will then move all the references to that point. To aid in formatting the collected references, *refer* writes the references preceded by the line

```
.]<
```

and followed by the line

```
.]>
```

to invoke special macros before and after the references.

Another possible option to *refer* is the **−s** option to specify sorting of references. The default, of course, is to list references in the order presented. The **−s** option implies the **−e** option, and thus requires a

```
.[
$LIST$
.]
```

entry to call out the reference list. The **−s** option may be followed by a string of letters, numbers, and '+' signs indicating how the references are to be sorted. The sort is done using the fields whose key-letters are in the string as sorting keys; the numbers indicate how many of the fields are to be considered, with '+' taken as a large number. Thus the default is **−sAD** meaning "Sort on senior author, then date." To sort on all authors and then title, specify **−sA+T**. And to sort on two authors and then the journal, write **−sA2J**.

Other options to *refer* change the signal or label inserted in the text for each reference. Normally these are just sequential numbers, and their exact placement (within brackets, as superscripts, etc.) is determined by the macro package. The **−l** option replaces reference numbers by strings composed of the senior author's last name, the date, and a disambiguating letter. If a number follows the **l** as in **−l3** only that many letters of the last name are used in the label string. To abbreviate the date as well the form **-l***m,n* shortens the last name to the first *m* letters and the date to the last *n* digits. For example, the option **−l3,2** would refer to the *eqn* paper (reference 3) by the signal *Ker75a*, since it is the first cited reference by Kernighan in 1975.

A user wishing to specify particular labels for a private bibliography may use the **−k** option. Specifying **−k***x* causes the field *x* to be used as a label. The default is **L**. If this field ends in **−**, that character is replaced by a sequence letter; otherwise the field is used exactly as given.

If none of the *refer*-produced signals are desired, the **−b** option entirely suppresses automatic text signals.

If the user wishes to override the *−ms* treatment of the reference signal (which is normally to enclose the number in brackets in *nroff* and make it a superscript in *troff*) this can be done easily. If the lines **.[** or **.]** contain anything following these characters, the remainders of these lines are used to surround the reference signal, instead of the default. Thus, for example, to say "See reference (2)." and avoid "See reference.^{2}" the input might appear

```
See reference
.[ (
imprecise citation ...
.]).
```

Note that blanks are significant in this construction. If a permanent change is desired in the style of reference signals, however, it is probably easier to redefine the strings **[.** and **.]** (which are used to bracket each signal) than to change each citation.

Although normally *refer* limits itself to retrieving the data for the reference, and leaves to a macro package the job of arranging that data as required by the local format, there are two special options for rearrangements that can not be done by macro packages. The **−c** option puts fields into all upper case (CAPS-SMALL CAPS in *troff* output). The key-letters indicated what information is to be translated to upper case follow the **c**, so that **−cAJ** means that authors' names and journals are to be in caps. The **−a** option writes the names of authors last name first, that is *A. D. Hall, Jr.* is written as *Hall, A. D. Jr.* The citation form of the *Journal of the ACM*, for example, would require both **−cA** and **−a** options. This produces authors' names in the style *KERNIGHAN, B. W. AND CHERRY, L. L.* for the previous example. The **−a** option may be followed by a number to indicate how many author names should be reversed; **−a1** (without any **−c** option) would produce *Kernighan, B. W. and L. L. Cherry*, for example.

Finally, there is also the previously-mentioned **−p** option to let the user specify a private file of references to be searched before the public files. Note that *refer* does not insist on a previously made index for these files. If a file is named which contains reference data but is not indexed, it will be searched (more slowly) by *refer* using *fgrep*. In this way it is easy for users to keep small files of new references, which can later be added to the public data bases.

Refer — A Bibliography System

Bill Tuthill

Computing Services
University of California
Berkeley, CA 94720

ABSTRACT

Refer is a bibliography system that supports data entry, indexing, retrieval, sorting, runoff, convenient citations, and footnote or endnote numbering. This document assumes you know how to use some Unix editor, and that you are familiar with the **nroff/troff** text formatters.

The **refer** program is a preprocessor for **nroff/troff,** like **eqn** and **tbl,** except that it is used for literature citations, rather than for equations and tables. Given incomplete but sufficiently precise citations, **refer** finds references in a bibliographic database. The complete references are formatted as footnotes, numbered, and placed either at the bottom of the page, or at the end of a chapter.

A number of ancillary programs make **refer** easier to use. The **addbib** program is for creating and extending the bibliographic database; **sortbib** sorts the bibliography by author and date, or other selected criteria; and **roffbib** runs off the entire database, formatting it not as footnotes, but as a bibliography or annotated bibliography.

Once a full bibliography has been created, access time can be improved by making an index to the references with **indxbib**. Then, the **lookbib** program can be used to quickly retrieve individual citations or groups of citations. Creating this inverted index will speed up **refer,** and **lookbib** will allow you to verify that a citation is sufficiently precise to deliver just one reference.

July 27, 1983

Table of Contents

Refer — A Bibliography System

Bill Tuthill

Computing Services
University of California
Berkeley, CA 94720

Introduction

Taken together, the **refer** programs constitute a database system for use with variable-length information. To distinguish various types of bibliographic material, the system uses labels composed of upper case letters, preceded by a percent sign and followed by a space. For example, one document might be given this entry:

```
%A  Joel Kies
%T  Document Formatting on Unix Using the -ms Macros
%I  Computing Services
%C  Berkeley
%D  1980
```

Each line is called a field, and lines grouped together are called a record; records are separated from each other by a blank line. Bibliographic information follows the labels, containing data to be used by the **refer** system. The order of fields is not important, except that authors should be entered in the same order as they are listed on the document. Fields can be as long as necessary, and may even be continued on the following line(s).

The labels are meaningful to **nroff/troff** macros, and, with a few exceptions, the **refer** program itself does not pay attention to them. This implies that you can change the label codes, if you also change the macros used by **nroff/troff**. The macro package takes care of details like proper ordering, underlining the book title or journal name, and quoting the article's title. Here are the labels used by **refer,** with an indication of what they represent:

%H	Header commentary, printed before reference
%A	Author's name
%Q	Corporate or foreign author (unreversed)
%T	Title of article or book
%S	Series title
%J	Journal containing article
%B	Book containing article
%R	Report, paper, or thesis (for unpublished material)
%V	Volume
%N	Number within volume
%E	Editor of book containing article
%P	Page number(s)
%I	Issuer (publisher)
%C	City where published
%D	Date of publication
%O	Other commentary, printed at end of reference
%K	Keywords used to locate reference
%L	Label used by −k option of **refer**
%X	Abstract (used by **roffbib,** not by **refer**)

Only relevant fields should be supplied. Except for %A, each field should be given only once; in the case of multiple authors, the senior author should come first. The %Q is for organizational authors, or authors with Japanese or Arabic names, in which cases the order of names should be preserved. Books should be labeled with the %T, not with the %B, which is reserved for books containing articles. The %J and %B fields should never appear together, although if they do, the %J will override the %B. If there is no author, just an editor, it is best to type the editor in the %A field, as in this example:

```
%A  Bertrand Bronson, ed.
```

The %E field is used for the editor of a book (%B) containing an article, which has its own author. For unpublished material such as theses, use the %R field; the title in the %T field will be quoted, but the contents of the %R field will not be underlined. Unlike other fields, %H, %O, and %X should contain their own punctuation. Here is a modest example:

```
%A  Mike E. Lesk
%T  Some Applications of Inverted Indexes on the Unix System
%B  Unix Programmer's Manual
%I  Bell Laboratories
%C  Murray Hill, NJ
%D  1978
%V  2a
%K  refer mkey inv hunt
%X  Difficult to read paper that dwells on indexing strategies,
giving little practical advice about using \fBrefer\fP.
```

Note that the author's name is given in normal order, without inverting the surname; inversion is done automatically, except when %Q is used instead of %A. We use %X rather than %O for the commentary because we do not want the comment printed all the time. The %O and %H fields are printed by both **refer** and **roffbib**; the %X field is printed only by **roffbib**, as a detached annotation paragraph.

Data Entry with Addbib

The **addbib** program is for creating and extending bibliographic databases. You must give it the filename of your bibliography:

```
% addbib database
```

Every time you enter **addbib**, it asks if you want instructions. To get them, type **y**; to skip them, type RETURN. **Addbib** prompts for various fields, reads from the keyboard, and writes records containing the **refer** codes to the database. After finishing a field entry, you should end it by typing RETURN. If a field is too long to fit on a line, type a backslash (\) at the end of the line, and you will be able to continue on the following line. Note: the backslash works in this capacity only inside **addbib**.

A field will not be written to the database if nothing is entered into it. Typing a minus sign as the first character of any field will cause **addbib** to back up one field at a time. Backing up is the best way to add multiple authors, and it really helps if you forget to add something important. Fields not contained in the prompting skeleton may be entered by typing a backslash as the last character before RETURN. The following line will be sent verbatim to the database and **addbib** will resume with the next field. This is identical to the procedure for dealing with long fields, but with new fields, don't forget the % key-letter.

Finally, you will be asked for an abstract (or annotation), which will be preserved as the %X field. Type in as many lines as you need, and end with a control-D (hold down the CTRL button, then press the "d" key). This prompting for an abstract can be suppressed with the **−a** command line option.

After one bibliographic record has been completed, **addbib** will ask if you want to continue. If you do, type RETURN; to quit, type **q** or **n** (quit or no). It is also possible to use one of the system editors to correct mistakes made while entering data. After the "Continue?" prompt, type any of the following: **edit, ex, vi,** or **ed** — you will be placed inside the corresponding editor, and returned to **addbib** afterwards, from where you can either quit or add more data.

If the prompts normally supplied by **addbib** are not enough, are in the wrong order, or are too numerous, you can redefine the skeleton by constructing a promptfile. Create some file, to be named after the −p command line option. Place the prompts you want on the left side, followed by a single TAB (control-I), then the **refer** code that is to appear in the bibliographic database. **Addbib** will send the left side to the screen, and the right side, along with data entered, to the database.

Printing the Bibliography

Sortbib is for sorting the bibliography by author (%A) and date (%D), or by data in other fields. It is quite useful for producing bibliographies and annotated bibliographies, which are seldom entered in strict alphabetical order. It takes as arguments the names of up to 16 bibliography files, and sends the sorted records to standard output (the terminal screen), which may be redirected through a pipe or into a file.

The −s*KEYS* flag to **sortbib** will sort by fields whose key-letters are in the *KEYS* string, rather than merely by author and date. Key-letters in *KEYS* may be followed by a '+' to indicate that all such fields are to be used. The default is to sort by senior author and date (printing the senior author last name first), but −sA+D will sort by all authors and then date, and −sATD will sort on senior author, then title, and then date.

Roffbib is for running off the (probably sorted) bibliography. It can handle annotated bibliographies — annotations are entered in the %X (abstract) field. **Roffbib** is a shell script that calls **refer −B** and **nroff −mbib**. It uses the macro definitions that reside in /usr/lib/tmac/tmac.bib, which you can redefine if you know **nroff** and **troff**. Note that **refer** will print the %H and %O commentaries, but will ignore abstracts in the %X field; **roffbib** will print both fields, unless annotations are suppressed with the −x option.

The following command sequence will lineprint the entire bibliography, organized alphabetically by author and date:

```
% sortbib database | roffbib | lpr
```

This is a good way to proofread the bibliography, or to produce a stand-alone bibliography at the end of a paper. Incidentally, **roffbib** accepts all flags used with **nroff**. For example:

```
% sortbib database | roffbib −Tdtc −s1
```

will make accent marks work on a DTC daisy-wheel printer, and stop at the bottom of every page for changing paper. The −n and −o flags may also be quite useful, to start page numbering at a selected point, or to produce only specific pages.

Roffbib understands four command-line number registers, which are something like the two-letter number registers in −ms. The −rN1 argument will number references beginning at one (1); use another number to start somewhere besides one. The −rV2 flag will double-space the entire bibliography, while −rV1 will double-space the references, but single-space the annotation paragraphs. Finally, specifying −rL6i changes the line length from 6.5 inches to 6 inches, and saying −rO1i sets the page offset to one inch, instead of zero. (That's a capital O after −r, not a zero.)

Citing Papers with Refer

The **refer** program normally copies input to output, except when it encounters an item of the form:

```
.[
partial citation
.]
```

The partial citation may be just an author's name and a date, or perhaps a title and a keyword, or maybe just a document number. **Refer** looks up the citation in the bibliographic database, and transforms it into a full, properly formatted reference. If the partial citation does not correctly identify a single work (either finding nothing, or more than one reference), a diagnostic message is given. If nothing is found, it will say "No such paper." If more than one reference is found, it will say "Too many hits." Other diagnostic messages can be quite cryptic; if you are in doubt, use **checknr** to verify that all your .['s have matching .]'s.

When everything goes well, the reference will be brought in from the database, numbered, and placed at the bottom of the page. This citation,[1] for example, was produced by:

```
This citation,
.[
lesk inverted indexes
.]
for example, was produced by
```

The .[and .] markers, in essence, replace the .FS and .FE of the −ms macros, and also provide a numbering mechanism. Footnote numbers will be bracketed on the the lineprinter, but superscripted on daisy-wheel terminals and in **troff**. In the reference itself, articles will be quoted, and books and journals will be underlined in **nroff,** and italicized in **troff.**

Sometimes you need to cite a specific page number along with more general bibliographic material. You may have, for instance, a single document that you refer to several times, each time giving a different page citation. This is how you could get "p. 10" in the reference:

```
.[
kies document formatting
%P  10
.]
```

The first line, a partial citation, will find the reference in your bibliography. The second line will insert the page number into the final citation. Ranges of pages may be specified as "%P 56-78".

When the time comes to run off a paper, you will need to have two files: the bibliographic database, and the paper to format. Use a command line something like one of these:

```
% refer −p database paper | nroff −ms
% refer −p database paper | tbl | nroff −ms
% refer −p database paper | tbl | neqn | nroff −ms
```

If other preprocessors are used, **refer** should precede **tbl,** which must in turn precede **eqn** or **neqn**. The −p option specifies a "private" database, which most bibliographies are.

Refer's Command-line Options

Many people like to place references at the end of a chapter, rather than at the bottom of the page. The −e option will accumulate references until a macro sequence of the form

```
.[
$LIST$
.]
```

is encountered (or until the end of file). **Refer** will then write out all references collected up to that point, collapsing identical references. Warning: there is a limit (currently 200) on the number of references that can be accumulated at one time.

It is also possible to sort references that appear at the end of text. The −s*KEYS* flag will sort references by fields whose key-letters are in the *KEYS* string, and permute reference numbers in the text accordingly. It is unnecessary to use −e with it, since −s implies −e. Key-letters in *KEYS* may

[1] Mike E. Lesk, "Some Applications of Inverted Indexes on the Unix System," *Unix Programmer's Manual*, vol. 2a, Bell Laboratories, Murray Hill, NJ, 1978.

be followed by a '+' to indicate that all such fields are to be used. The default is to sort by senior author and date, but −sA+D will sort on all authors and then date, and −sA+T will sort by authors and then title.

Refer can also make citations in what is known as the Social or Natural Sciences format. Instead of numbering references, the −l (letter ell) flag makes labels from the senior author's last name and the year of publication. For example, a reference to the paper on Inverted Indexes cited above might appear as [Lesk1978a]. It is possible to control the number of characters in the last name, and the number of digits in the date. For instance, the command line argument −l6,2 might produce a reference such as [Kernig78c].

Some bibliography standards shun both footnote numbers and labels composed of author and date, requiring some keyword to identify the reference. The −k flag indicates that, instead of numbering references, key labels specified on the %L line should be used to mark references.

The −n flag means to not search the default reference file, located in /usr/dict/papers/Rv7man. Using this flag may make **refer** marginally faster. The −a*n* flag will reverse the first *n* author names, printing Jones, J. A. instead of J. A. Jones. Often −a1 is enough; this will reverse the names of only the senior author. In some versions of **refer** there is also the −f flag to set the footnote number to some predetermined value; for example, −f23 would start numbering with footnote 23.

Making an Index

Once your database is large and relatively stable, it is a good idea to make an index to it, so that references can be found quickly and efficiently. The **indxbib** program makes an inverted index to the bibliographic database (this program is called **pubindex** in the Bell Labs manual). An inverted index could be compared to the thumb cuts of a dictionary — instead of going all the way through your bibliography, programs can move to the exact location where a citation is found.

Indxbib itself takes a while to run, and you will need sufficient disk space to store the indexes. But once it has been run, access time will improve dramatically. Furthermore, large databases of several million characters can be indexed with no problem. The program is exceedingly simple to use:

```
% indxbib database
```

Be aware that changing your database will require that you run **indxbib** over again. If you don't, you may fail to find a reference that really is in the database.

Once you have built an inverted index, you can use **lookbib** to find references in the database. **Lookbib** cannot be used until you have run **indxbib**. When editing a paper, **lookbib** is very useful to make sure that a citation can be found as specified. It takes one argument, the name of the bibliography, and then reads partial citations from the terminal, returning references that match, or nothing if none match. Its prompt is the greater-than sign.

```
% lookbib database
> lesk inverted indexes
%A  Mike E. Lesk
%T  Some Applications of Inverted Indexes on the Unix System
%J  Unix Programmer's Manual
%I  Bell Laboratories
%C  Murray Hill, NJ
%D  1978
%V  2a
%X  Difficult to read paper that dwells on indexing strategies,
giving little practical advice about using \fBrefer\fP.
>
```

If more than one reference comes back, you will have to give a more precise citation for **refer**. Experiment until you find something that works; remember that it is harmless to overspecify. To get out of the **lookbib** program, type a control-D alone on a line; **lookbib** then exits with an "EOT" message.

Lookbib can also be used to extract groups of related citations. For example, to find all the papers by Brian Kernighan found in the system database, and send the output to a file, type:

```
% lookbib /usr/dict/papers/Ind > kern.refs
> kernighan
> EOT
% cat kern.refs
```

Your file, "kern.refs", will be full of references. A similar procedure can be used to pull out all papers of some date, all papers from a given journal, all papers containing a certain group of keywords, etc.

Refer Bugs and Some Solutions

The **refer** program will mess up if there are blanks at the end of lines, especially the %A author line. **Addbib** carefully removes trailing blanks, but they may creep in again during editing. Use an editor command — g/ *$/s/// — to remove trailing blanks from your bibliography.

Having bibliographic fields passed through as string definitions implies that interpolated strings (such as accent marks) must have two backslashes, so they can pass through copy mode intact. For instance, the word "téléphone" would have to be represented:

```
te\\*'le\\*'phone
```

in order to come out correctly. In the %X field, by contrast, you will have to use single backslashes instead. This is because the %X field is not passed through as a string, but as the body of a paragraph macro.

Another problem arises from authors with foreign names. When a name like "Valéry Giscard d'Estaing" is turned around by the −a option of **refer**, it will appear as "d'Estaing, Valéry Giscard," rather than as "Giscard d'Estaing, Valéry." To prevent this, enter names as follows:

```
%A  Vale\\*'ry  Giscard\0d'Estaing
%A  Alexander  Csoma\0de\0Ko\\*:ro\\*:s
```

(The second is the name of a famous Hungarian linguist.) The backslash-zero is an **nroff/troff** request meaning to insert a digit-width space. It will protect against faulty name reversal, and also against mis-sorting.

Footnote numbers are placed at the end of the line before the .[macro. This line should be a line of text, not a macro. As an example, if the line before the .[is a .R macro, then the .R will eat the footnote number. (The .R is an −ms request meaning change to Roman font.) In cases where the font needs changing, it is necessary to do the following:

```
\fIet al.\fR
.[
awk  aho  kernighan  weinberger
.]
```

Now the reference will be to Aho *et al.*[2] The \fI changes to italics, and the \fR changes back to Roman font. Both these requests are **nroff/troff** requests, not part of −ms. If and when a footnote number is added after this sequence, it will indeed appear in the output.

[2] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, "Awk — A Pattern Scanning and Processing Language," *Unix Programmer's Manual*, vol. 2a, Bell Laboratories, Murray Hill, NJ, 1978.

Internal Details of Refer

You have already read everything you need to know in order to use the **refer** bibliography system. The remaining sections are provided only for extra information, and in case you need to change the way **refer** works.

The output of **refer** is a stream of string definitions, one for each field in a reference. To create string names, percent signs are simply changed to an open bracket, and an [F string is added, containing the footnote number. The %X, %Y and %Z fields are ignored; however, the **annobib** program changes the %X to an .AP (annotation paragraph) macro. The citation used above yields this intermediate output:

```
.ds [F 1
.]-
.ds [A Mike E. Lesk
.ds [T Some Applications of Inverted Indexes on the Unix System
.ds [J Unix Programmer's Manual
.ds [I Bell Laboratories
.ds [C Murray Hill, NJ
.ds [D 1978
.ds [V 2a
.nr [T 0
.nr [A 0
.nr [O 0
.][ 1 journal-article
```

These string definitions are sent to **nroff,** which can use the −ms macros defined in /usr/lib/mx/tmac.xref to take care of formatting things properly. The initializing macro **.]−** precedes the string definitions, and the labeled macro **.][** follows. These are changed from the input **.[** and **.]** so that running a file twice through **refer** is harmless.

The **.][** macro, used to print the reference, is given a type-number argument, which is a numeric label indicating the type of reference involved. Here is a list of the various kinds of references:

Field	Value	Kind of Reference
%J	1	Journal Article
%B	3	Article in Book
%R %G	4	Report, Government Report
%I	2	Book
%M	5	Bell Labs Memorandum (undefined)
none	0	Other

The order listed above is indicative of the precedence of the various fields. In other words, a reference that has both the %J and %B fields will be classified as a journal article. If none of the fields listed is present, then the reference will be classified as "other."

The footnote number is flagged in the text with the following sequence, where *number* is the footnote number:

*([.*number**(.]

The *([. and *(.] stand for bracketing or superscripting. In **nroff** with low-resolution devices such as the lpr and a crt, footnote numbers will be bracketed. In **troff,** or on daisy-wheel printers, footnote numbers will be superscripted. Punctuation normally comes before the reference number; this can be changed by using the −P (postpunctuation) option of **refer.**

In some cases, it is necessary to override certain fields in a reference. For instance, each time a work is cited, you may want to specify different page numbers, and you may want to change certain fields. This citation will find the Lesk reference, but will add specific page numbers to the output, even though no page numbers appeared in the original reference.

```
        .[
        lesk  inverted  indexes
        %P   7-13
        %I   Computing Services
        %O   UNX 12.2.2.
        .]
```

The %I line will also override any previous publisher information, and the %O line will append some commentary. The **refer** program simply adds the new %P, %I, and %O strings to the output, and later strings definitions cancel earlier ones.

It is also possible to insert an entire citation that does not appear in the bibliographic database. This reference, for example, could be added as follows:

```
        .[
        %A   Brian Kernighan
        %T   A Troff Tutorial
        %I   Bell Laboratories
        %D   1978
        .]
```

This will cause **refer** to interpret the fields exactly as given, without searching the bibliographic database. This practice is not recommended, however, because it's better to add new references to the database, so they can be used again later.

If you want to change the way footnote numbers are printed, signals can be given on the **.[** and **.]** lines. For example, to say "See reference (2)," the citation should appear as:

```
        See reference
        .[(
        partial citation
        .]),
```

Note that blanks are significant on these signal lines. If a permanent change in the footnote format is desired, it's best to redefine the **[.** and **.]** strings.

Changing the Refer Macros

This section is provided for those who wish to rewrite or modify the **refer** macros. This is necessary in order to make output correspond to specific journal requirements, or departmental standards. First there is an explanation of how new macros can be substituted for the old ones. Then several alterations are given as examples. Finally, there is an annotated copy of the **refer** macros used by **roffbib**.

The **refer** macros for **nroff/troff** supplied by the **−ms** macro package reside in /usr/lib/mx/tmac.xref; they are reference macros, for producing footnotes or endnotes. The **refer** macros used by **roffbib**, on the other hand, reside in /usr/lib/tmac/tmac.bib; they are for producing a stand-alone bibliography.

To change the macros used by **roffbib**, you will need to get your own version of this shell script into the directory where you are working. These two commands will get you a copy of **roffbib** and the macros it uses: †

```
        % cp /usr/lib/tmac/tmac.bib bibmac
```

You can proceed to change bibmac as much as you like. Then when you use **roffbib**, you should specify your own version of the macros, which will be substituted for the normal ones

```
        % roffbib −m bibmac filename
```

where *filename* is the name of your bibliography file. Make sure there's a space between −m and **bib-mac**.

If you want to modify the **refer** macros for use with **nroff** and the −ms macros, you will need to get a copy of "tmac.xref":

```
% cp /usr/lib/ms/s.ref refmac
```

These macros are much like "bibmac", except they have .FS and .FE requests, to be used in conjunction with the −ms macros, rather than independently defined .XP and .AP requests. Now you can put this line at the top of the paper to be formatted:

```
.so refmac
```

Your new **refer** macros will override the definitions previously read in by the −ms package. This method works only if "refmac" is in the working directory.

Suppose you didn't like the way dates are printed, and wanted them to be parenthesized, with no comma before. There are five identical lines you will have to change. The first line below is the old way, while the second is the new way:

```
.if !"\\*([D"" , \\*([D\c
.if !"\\*([D"" \& (\\*([D)\c
```

In the first line, there is a comma and a space, but no parentheses. The "\c" at the end of each line indicates to **nroff** that it should continue, leaving no extra space in the output. The "\&" in the second line is the do-nothing character; when followed by a space, a space is sent to the output.

If you need to format a reference in the style favored by the Modern Language Association or Chicago University Press, in the form (city: publisher, date), then you will have to change the middle of the book macro [2 as follows:

```
\& (\c
.if !"\\*([C"" \\*([C:
\\*([I\c
.if !"\\*([D"" , \\*([D\c
)\c
```

This would print (Berkeley: Computing Services, 1982) if all three strings were present. The first line prints a space and a parenthesis; the second prints the city (and a colon) if present; the third always prints the publisher (books must have a publisher, or else they're classified as other); the fourth line prints a comma and the date if present; and the fifth line closes the parentheses. You would need to make similar changes to the other macros as well.

Acknowledgements

Mike Lesk of Bell Laboratories wrote the original **refer** software, including the indexing programs. Al Stangenberger of the Forestry Department wrote the first version of **addbib**, then called **bibin**. Greg Shenaut of the Linguistics Department wrote the original versions of **sortbib** and **roffbib**. All these contributions are greatly appreciated.

Writing Tools - The STYLE and DICTION Programs

L. L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

W. Vesterman

Livingston College
Rutgers University

ABSTRACT

Text processing systems are now in heavy use in many companies to format documents. With many documents stored on line, it has become possible to use computers to study writing style itself and to help writers produce better written and more readable prose. The system of programs described here is an initial step toward such help. It includes programs and a data base designed to produce a stylistic profile of writing at the word and sentence level. The system measures readability, sentence and word length, sentence type, word usage, and sentence openers. It also locates common examples of wordy phrasing and bad diction. The system is useful for evaluating a document's style, locating sentences that may be difficult to read or excessively wordy, and determining a particular writer's style over several documents.

November 22, 1980

Writing Tools - The STYLE and DICTION Programs

L. L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

W. Vesterman

Livingston College
Rutgers University

1. Introduction

Computers have become important in the document preparation process, with programs to check for spelling errors and to format documents. As the amount of text stored on line increases, it becomes feasible and attractive to study writing style and to attempt to help the writer in producing readable documents. The system of writing tools described here is a first step toward such help. The system includes programs and a data base to analyze writing style at the word and sentence level. We use the term ''style'' in this paper to describe the results of a writer's particular choices among individual words and sentence forms. Although many judgements of style are subjective, particularly those of word choice, there are some objective measures that experts agree lead to good style. Three programs have been written to measure some of the objectively definable characteristics of writing style and to identify some commonly misused or unnecessary phrases. Although a document that conforms to the stylistic rules is not guaranteed to be coherent and readable, one that violates all of the rules is likely to be difficult or tedious to read. The program STYLE calculates readability, sentence length variability, sentence type, word usage and sentence openers at a rate of about 400 words per second on a PDP11/70 running the UNIX† Operating System. It assumes that the sentences are well-formed, i. e. that each sentence has a verb and that the subject and verb agree in number. DICTION identifies phrases that are either bad usage or unnecessarily wordy. EXPLAIN acts as a thesaurus for the phrases found by DICTION. Sections 2, 3, and 4 describe the programs; Section 5 gives the results on a cross-section of technical documents; Section 6 discusses accuracy and problems; Section 7 gives implementation details.

2. STYLE

The program STYLE reads a document and prints a summary of readability indices, sentence length and type, word usage, and sentence openers. It may also be used to locate all sentences in a document longer than a given length, of readability index higher than a given number, those containing a passive verb, or those beginning with an expletive. STYLE is based on the system for finding English word classes or parts of speech, PARTS [1]. PARTS is a set of programs that uses a small dictionary (about 350 words) and suffix rules to partially assign word classes to English text. It then uses experimentally derived rules of word order to assign word classes to all words in the text with an accuracy of about 95%. Because PARTS uses only a small dictionary and general rules, it works on text about any subject, from physics to psychology. Style measures have been built into the output phase of the programs that make up PARTS. Some of the measures are simple counters of the word classes found by PARTS; many are more complicated. For example, the verb count is the total number of verb phrases. This includes phrases like:

†UNIX is a Trademark of Bell Laboratories.

has been going
was only going
to go

each of which each counts as one verb. Figure 1 shows the output of STYLE run on a paper by Kernighan and Mashey about the UNIX programming environment [2].

```
programming environment
readability grades:
                        (Kincaid) 12.3  (auto) 12.8  (Coleman-Liau) 11.8  (Flesch) 13.5 (46.3)
sentence info:
                        no. sent 335 no. wds 7419
                        av sent leng 22.1 av word leng 4.91
                        no. questions 0 no. imperatives 0
                        no. nonfunc wds 4362  58.8%   av leng 6.38
                        short sent (<17) 35% (118) long sent (>32)  16% (55)
                        longest sent 82 wds at sent 174; shortest sent 1 wds at sent 117
sentence types:
                        simple  34% (114) complex  32% (108)
                        compound  12% (41) compound-complex  21% (72)
word usage:
                        verb types as % of total verbs
                        tobe  45% (373) aux  16% (133) inf  14% (114)
                        passives as % of non-inf verbs  20% (144)
                        types as % of total
                        prep 10.8% (804) conj 3.5% (262) adv 4.8% (354)
                        noun 26.7% (1983) adj 18.7% (1388) pron 5.3% (393)
                        nominalizations   2 % (155)
sentence beginnings:
                        subject opener: noun (63) pron (43) pos (0) adj (58) art (62) tot  67%
                        prep  12% (39) adv   9% (31)
                        verb   0% (1)  sub_conj   6% (20) conj   1% (5)
                        expletives   4% (13)
```

Figure 1

As the example shows, STYLE output is in five parts. After a brief discussion of sentences, we will describe the parts in order.

2.1. What is a sentence?

Readers of documents have little trouble deciding where the sentences end. People don't even have to stop and think about uses of the character "." in constructions like 1.25, A. J. Jones, Ph.D., i. e., or etc. . When a computer reads a document, finding the end of sentences is not as easy. First we must throw away the printer's marks and formatting commands that litter the text in computer form. Then STYLE defines a sentence as a string of words ending in one of:

. ! ? /.

The end marker "/." may be used to indicate an imperative sentence. Imperative sentences that are not so marked are not identified as imperative. STYLE properly handles numbers with embedded decimal points and commas, strings of letters and numbers with embedded decimal points used for naming computer file names, and the common abbreviations listed in Appendix

1. Numbers that end sentences, like the preceding sentence, cause a sentence break if the next word begins with a capital letter. Initials only cause a sentence break if the next word begins with a capital and is found in the dictionary of function words used by PARTS. So the string

J. D. JONES

does not cause a break, but the string

... system H. The ...

does. With these rules most sentences are broken at the proper place, although occasionally either two sentences are called one or a fragment is called a sentence. More on this later.

2.2. Readability Grades

The first section of STYLE output consists of four readability indices. As Klare points out in [3] readability indices may be used to estimate the reading skills needed by the reader to understand a document. The readability indices reported by STYLE are based on measures of sentence and word lengths. Although the indices may not measure whether the document is coherent and well organized, experience has shown that high indices seem to be indicators of stylistic difficulty. Documents with short sentences and short words have low scores; those with long sentences and many polysyllabic words have high scores. The 4 formulae reported are Kincaid Formula [4], Automated Readability Index [5], Coleman-Liau Formula [6] and a normalized version of Flesch Reading Ease Score [7]. The formulae differ because they were experimentally derived using different texts and subject groups. We will discuss each of the formulae briefly; for a more detailed discussion the reader should see [3].

The Kincaid Formula, given by:

$$Reading\_Grade = 11.8 * syl\_per\_wd + .39 * wds\_per\_sent - 15.59$$

was based on Navy training manuals that ranged in difficulty from 5.5 to 16.3 in reading grade level. The score reported by this formula tends to be in the mid-range of the 4 scores. Because it is based on adult training manuals rather than school book text, this formula is probably the best one to apply to technical documents.

The Automated Readability Index (ARI), based on text from grades 0 to 7, was derived to be easy to automate. The formula is:

$$Reading\_Grade = 4.71 * let\_per\_wd + .5 * wds\_per\_sent - 21.43$$

ARI tends to produce scores that are higher than Kincaid and Coleman-Liau but are usually slightly lower than Flesch.

The Coleman-Liau Formula, based on text ranging in difficulty from .4 to 16.3, is:

$$Reading\_Grade = 5.89 * let\_per\_wd - .3 * sent\_per\_100\_wds - 15.8$$

Of the four formulae this one usually gives the lowest grade when applied to technical documents.

The last formula, the Flesch Reading Ease Score, is based on grade school text covering grades 3 to 12. The formula, given by:

$$Reading\_Score = 206.835 - 84.6 * syl\_per\_wd - 1.015 * wds\_per\_sent$$

is usually reported in the range 0 (very difficult) to 100 (very easy). The score reported by STYLE is scaled to be comparable to the other formulas, except that the maximum grade level reported is set to 17. The Flesch score is usually the highest of the 4 scores on technical documents.

Coke [8] found that the Kincaid Formula is probably the best predictor for technical documents; both ARI and Flesch tend to overestimate the difficulty; Coleman-Liau tend to underestimate. On text in the range of grades 7 to 9 the four formulas tend to be about the same. On easy text the Coleman-Liau formula is probably preferred since it is reasonably accurate at the lower grades and it is safer to present text that is a little too easy than a little too hard.

If a document has particularly difficult technical content, especially if it includes a lot of mathematics, it is probably best to make the text very easy to read, i.e. a lower readability index by shortening the sentences and words. This will allow the reader to concentrate on the technical content and not the long sentences. The user should remember that these indices are estimators; they should not be taken as absolute numbers. STYLE called with "−r number" will print all sentences with an Automated Readability Index equal to or greater than "number".

2.3. Sentence length and structure

The next two sections of STYLE output deal with sentence length and structure. Almost all books on writing style or effective writing emphasize the importance of variety in sentence length and structure for good writing. Ewing's first rule in discussing style in the book *Writing for Results* [9] is:

> "Vary the sentence structure and length of your sentences."

Leggett, Mead and Charvat break this rule into 3 in *Prentice-Hall Handbook for Writers* [10] as follows:

> "34a. Avoid the overuse of short simple sentences."
> "34b. Avoid the overuse of long compound sentences."
> "34c. Use various sentence structures to avoid monotony and increase effectiveness."

Although experts agree that these rules are important, not all writers follow them. Sample technical documents have been found with almost no sentence length or type variability. One document had 90% of its sentences about the same length as the average; another was made up almost entirely of simple sentences (80%).

The output sections labeled "sentence info" and "sentence types" give both length and structure measures. STYLE reports on the number and average length of both sentences and words, and number of questions and imperative sentences (those ending in "/."). The measures of non-function words are an attempt to look at the content words in the document. In English non-function words are nouns, adjectives, adverbs, and non-auxiliary verbs; function words are prepositions, conjunctions, articles, and auxiliary verbs. Since most function words are short, they tend to lower the average word length. The average length of non-function words may be a more useful measure for comparing word choice of different writers than the total average word length. The percentages of short and long sentences measure sentence length variability. Short sentences are those at least 5 words less than the average; long sentences are those at least 10 words longer than the average. Last in the sentence information section is the length and location of the longest and shortest sentences. If the flag "−l number" is used, STYLE will print all sentences longer than "number".

Because of the difficulties in dealing with the many uses of commas and conjunctions in English, sentence type definitions vary slightly from those of standard textbooks, but still measure the same constructional activity.

1. A simple sentence has one verb and no dependent clause.
2. A complex sentence has one independent clause and one dependent clause, each with one verb. Complex sentences are found by identifying sentences that contain either a subordinate conjunction or a clause beginning with words like "that" or "who". The preceding sentence has such a clause.
3. A compound sentence has more than one verb and no dependent clause. Sentences joined by ";" are also counted as compound.
4. A compound-complex sentence has either several dependent clauses or one dependent clause and a compound verb in either the dependent or independent clause.

Even using these broader definitions, simple sentences dominate many of the technical documents that have been tested, but the example in Figure 1 shows variety in both sentence structure and sentence length.

2.4. Word Usage

The word usage measures are an attempt to identify some other constructional features of writing style. There are many different ways in English to say the same thing. The constructions differ from one another in the form of the words used. The following sentences all convey approximately the same meaning but differ in word usage:

> The cxio program is used to perform all communication between the systems.
> The cxio program performs all communications between the systems.
> The cxio program is used to communicate between the systems.
> The cxio program communicates between the systems.
> All communication between the systems is performed by the cxio program.

The distribution of the parts of speech and verb constructions helps identify overuse of particular constructions. Although the measures used by STYLE are crude, they do point out problem areas. For each category, STYLE reports a percentage and a raw count. In addition to looking at the percentage, the user may find it useful to compare the raw count with the number of sentences. If, for example, the number of infinitives is almost equal to the number of sentences, then many of the sentences in the document are constructed like the first and third in the preceding example. The user may want to transform some of these sentences into another form. Some of the implications of the word usage measures are discussed below.

Verbs are measured in several different ways to try to determine what types of verb constructions are most frequent in the document. Technical writing tends to contain many passive verb constructions and other usage of the verb "to be". The category of verbs labeled "tobe" measures both passives and sentences of the form:

> *subject tobe predicate*

In counting verbs, whole verb phrases are counted as one verb. Verb phrases containing auxiliary verbs are counted in the category "aux". The verb phrases counted here are those whose tense is not simple present or simple past. It might eventually be useful to do more detailed measures of verb tense or mood. Infinitives are listed as "inf". The percentages reported for these three categories are based on the total number of verb phrases found. These categories are not mutually exclusive; they cannot be added, since, for example, "to be going" counts as both "tobe" and "inf". Use of these three types of verb constructions varies significantly among authors.

STYLE reports passive verbs as a percentage of the finite verbs in the document. Most style books warn against the overuse of passive verbs. Coleman [11] has shown that sentences with active verbs are easier to learn than those with passive verbs. Although the inverted object-subject order of the passive voice seems to emphasize the object, Coleman's experiments showed that there is little difference in retention by word position. He also showed that the direct object of an active verb is retained better than the subject of a passive verb. These experiments support the advice of the style books suggesting that writers should try to use active verbs wherever possible. The flag "−p" causes STYLE to print all sentences containing passive verbs.

Pronouns add cohesiveness and connectivity to a document by providing back-reference. They are often a short-hand notation for something previously mentioned, and therefore connect the sentence containing the pronoun with the word to which the pronoun refers. Although there are other mechanisms for such connections, documents with no pronouns tend to be wordy and to have little connectivity.

Adverbs can provide transition between sentences and order in time and space. In performing these functions, adverbs, like pronouns, provide connectivity and cohesiveness.

Conjunctions provide parallelism in a document by connecting two or more equal units. These units may be whole sentences, verb phrases, nouns, adjectives, or prepositional phrases. The compound and compound-complex sentences reported under sentence type are parallel structures. Other uses of parallel structures are indicated by the degree that the number of conjunctions reported under word usage exceeds the compound sentence measures.

Nouns and Adjectives. A ratio of nouns to adjectives near unity may indicate the over-use of modifiers. Some technical writers qualify every noun with one or more adjectives. Qualifiers in phrases like "simple linear single-link network model" often lend more obscurity than precision to a text.

Nominalizations are verbs that are changed to nouns by adding one of the suffixes "ment", "ance", "ence", or "ion". Examples are accomplishment, admittance, adherence, and abbreviation. When a writer transforms a nominalized sentence to a non-nominalized sentence, she/he increases the effectiveness of the sentence in several ways. The noun becomes an active verb and frequently one complicated clause becomes two shorter clauses. For example,

> Their inclusion of this provision is admission of the importance of the system.
> When they included this provision, they admitted the importance of the system.

Coleman found that the transformed sentences were easier to learn, even when the transformation produced sentences that were slightly longer, provided the transformation broke one clause into two. Writers who find their document contains many nominalizations may want to transform some of the sentences to use active verbs.

2.5. Sentence openers

Another agreed upon principle of style is variety in sentence openers. Because STYLE determines the type of sentence opener by looking at the part of speech of the first word in the sentence, the sentences counted under the heading "subject opener" may not all really begin with the subject. However, a large percentage of sentences in this category still indicates lack of variety in sentence openers. Other sentence opener measures help the user determine if there are transitions between sentences and where the subordination occurs. Adverbs and conjunctions at the beginning of sentences are mechanisms for transition between sentences. A pronoun at the beginning shows a link to something previously mentioned and indicates connectivity.

The location of subordination can be determined by comparing the number of sentences that begin with a subordinator with the number of sentences with complex clauses. If few sentences start with subordinate conjunctions then the subordination is embedded or at the end of the complex sentences. For variety the writer may want to transform some sentences to have leading subordination.

The last category of openers, expletives, is commonly overworked in technical writing. Expletives are the words "it" and "there", usually with the verb "to be", in constructions where the subject follows the verb. For example,

> There are three streets used by the traffic.
> There are too many users on this system.

This construction tends to emphasize the object rather than the subject of the sentence. The flag "−e" will cause STYLE to print all sentences that begin with an expletive.

3. DICTION

The program DICTION prints all sentences in a document containing phrases that are either frequently misused or indicate wordiness. The program, an extension of Aho's FGREP [12] string matching program, takes as input a file of phrases or patterns to be matched and a file of text to be searched. A data base of about 450 phrases has been compiled as a default pattern file for DICTION. Before attempting to locate phrases, the program maps upper case letters to lower case and substitutes blanks for punctuation. Sentence boundaries were deemed less critical in DICTION than in STYLE, so abbreviations and other uses of the character "." are not treated specially. DICTION brackets all pattern matches in a sentence with the characters "[" "]" . Although many of the phrases in the default data base are correct in some contexts, in others they indicate wordiness. Some examples of the phrases and suggested alternatives are:

Phrase	Alternative
a large number of	many
arrive at a decision	decide
collect together	collect
for this reason	so
pertaining to	about
through the use of	by or with
utilize	use
with the exception of	except

Appendix 2 contains a complete list of the default file. Some of the entries are short forms of problem phrases. For example, the phrase "the fact" is found in all of the following and is sufficient to point out the wordiness to the user:

Phrase	Alternative
accounted for by the fact that	caused by
an example of this is the fact that	thus
based on the fact that	because
despite the fact that	although
due to the fact that	because
in light of the fact that	because
in view of the fact that	since
notwithstanding the fact that	although

Entries in Appendix 2 preceded by "~" are not matched. See Section 7 for details on the use of "~".

The user may supply her/his own pattern file with the flag "−f patfile". In this case the default file will be loaded first, followed by the user file. This mechanism allows users to suppress patterns contained in the default file or to include their own pet peeves that are not in the default file. The flag "−n" will exclude the default file altogether. In constructing a pattern file, blanks should be used before and after each phrase to avoid matching substrings in words. For example, to find all occurrences of the word "the", the pattern " the " should be used. The blanks cause only the word "the" to be matched and not the string "the" in words like there, other, and therefore. One side effect of surrounding the words with blanks is that when two phrases occur without intervening words, only the first will be matched.

4. EXPLAIN

The last program, EXPLAIN, is an interactive thesaurus for phrases found by DICTION. The user types one of the phrases bracketed by DICTION and EXPLAIN responds with suggested substitutions for the phrase that will improve the diction of the document.

Table 1
Text Statistics on 20 Technical Documents

	variable	minimum	maximum	mean	standard deviation
Readability	Kincaid	9.5	16.9	13.3	2.2
	automated	9.0	17.4	13.3	2.5
	Cole-Liau	10.0	16.0	12.7	1.8
	Flesch	8.9	17.0	14.4	2.2
sentence info.	av sent length	15.5	30.3	21.6	4.0
	av word length	4.61	5.63	5.08	.29
	av nonfunction length	5.72	7.30	6.52	.45
	short sent	23%	46%	33%	5.9
	long sent	7%	20%	14%	2.9
sentence types	simple	31%	71%	49%	11.4
	complex	19%	50%	33%	8.3
	compound	2%	14%	7%	3.3
	compound-complex	2%	19%	10%	4.8
verb types	tobe	26%	64%	44.7%	10.3
	auxiliary	10%	40%	21%	8.7
	infinitives	8%	24%	15.1%	4.8
	passives	12%	50%	29%	9.3
word usage	prepositions	10.1%	15.0%	12.3%	1.6
	conjunction	1.8%	4.8%	3.4%	.9
	adverbs	1.2%	5.0%	3.4%	1.0
	nouns	23.6%	31.6%	27.8%	1.7
	adjectives	15.4%	27.1%	21.1%	3.4
	pronouns	1.2%	8.4%	2.5%	1.1
	nominalizations	2%	5%	3.3%	.8
sentence openers	prepositions	6%	19%	12%	3.4
	adverbs	0%	20%	9%	4.6
	subject	56%	85%	70%	8.0
	verbs	0%	4%	1%	1.0
	subordinating conj	1%	12%	5%	2.7
	conjunctions	0%	4%	0%	1.5
	expletives	0%	6%	2%	1.7

5. Results

5.1. STYLE

To get baseline statistics and check the program's accuracy, we ran STYLE on 20 technical documents. There were a total of 3287 sentences in the sample. The shortest document was 67 sentences long; the longest 339 sentences. The documents covered a wide range of subject matter, including theoretical computing, physics, psychology, engineering, and affirmative action. Table 1 gives the range, median, and standard deviation of the various style measures. As you will note most of the measurements have a fairly wide range of values across the sample documents.

As a comparison, Table 2 gives the median results for two different technical authors, a sample of instructional material, and a sample of the Federalist Papers. The two authors show similar styles, although author 2 uses somewhat shorter sentences and longer words than author 1. Author 1 uses all types of sentences, while author 2 prefers simple and complex sentences, using few compound or compound-complex sentences. The other major difference in the styles of these authors is the location of subordination. Author 1 seems to prefer embedded or trailing subordination, while author 2 begins many sentences with the subordinate clause. The documents tested for both authors 1 and 2 were technical documents, written for a technical audience. The instructional documents, which are written for craftspeople, vary surprisingly little from the two technical samples. The sentences and words are a little longer, and they contain many passive and auxiliary verbs, few adverbs, and almost no pronouns. The instructional documents contain many imperative sentences, so there are many sentence with verb openers. The sample of Federalist Papers contrasts with the other samples in almost every way.

Table 2
Text Statistics on Single Authors

	variable	author 1	author 2	inst.	FED
readability	Kincaid	11.0	10.3	10.8	16.3
	automated	11.0	10.3	11.9	17.8
	Coleman-Liau	9.3	10.1	10.2	12.3
	Flesch	10.3	10.7	10.1	15.0
sentence info	av sent length	22.64	19.61	22.78	31.85
	av word length	4.47	4.66	4.65	4.95
	av nonfunction length	5.64	5.92	6.04	6.87
	short sent	35%	43%	35%	40%
	long sent	18%	15%	16%	21%
sentence types	simple	36%	43%	40%	31%
	complex	34%	41%	37%	34%
	compound	13%	7%	4%	10%
	compound-complex	16%	8%	14%	25%
verb type	tobe	42%	43%	45%	37%
	auxiliary	17%	19%	32%	32%
	infinitives	17%	15%	12%	21%
	passives	20%	19%	36%	20%
word usage	prepositions	10.0%	10.8%	12.3%	15.9%
	conjunctions	3.2%	2.4%	3.9%	3.4%
	adverbs	5.05%	4.6%	3.5%	3.7%
	nouns	27.7%	26.5%	29.1%	24.9%
	adjectives	17.0%	19.0%	15.4%	12.4%
	pronouns	5.3%	4.3%	2.1%	6.5%
	nominalizations	1%	2%	2%	3%
sentence openers	prepositions	11%	14%	6%	5%
	adverbs	9%	9%	6%	4%
	subject	65%	59%	54%	66%
	verb	3%	2%	14%	2%
	subordinating conj	8%	14%	11%	3%
	conjunction	1%	0%	0%	3%
	expletives	3%	3%	0%	3%

5.2. DICTION

In the few weeks that DICTION has been available to users about 35,000 sentences have been run with about 5,000 string matches. The authors using the program seem to make the suggested changes about 50-75% of the time. To date, almost 200 of the 450 strings in the default file have been matched. Although most of these phrases are valid and correct in some contexts, the 50-75% change rate seems to show that the phrases are used much more often than concise diction warrants.

6. Accuracy

6.1. Sentence Identification

The correctness of the STYLE output on the 20 document sample was checked in detail. STYLE misidentified 129 sentence fragments as sentences and incorrectly joined two or more sentences 75 times in the 3287 sentence sample. The problems were usually because of non-standard formatting commands, unknown abbreviations, or lists of non-sentences. An impossibly long sentence found as the longest sentence in the document usually is the result of a long list of non-sentences.

6.2. Sentence Types

Style correctly identified sentence type on 86.5% of the sentences in the sample. The type distribution of the sentences was 52.5% simple, 29.9% complex, 8.5% compound and 9% compound-complex. The program reported 49.5% simple, 31.9% complex, 8% compound and 10.4% compound-complex. Looking at the errors on the individual documents, the number of simple sentences was under-reported by about 4% and the complex and compound-complex were over-reported by 3% and 2%, respectively. The following matrix shows the programs output vs. the actual sentence type.

		Program Results			
		simple	complex	compound	comp-complex
Actual	simple	1566	132	49	17
Sentence	complex	47	892	6	65
Type	compound	40	6	207	23
	comp-complex	0	52	5	249

The system's inability to find imperative sentences seems to have little effect on most of the style statistics. A document with half of its sentences imperative was run, with and without the imperative end marker. The results were identical except for the expected errors of not finding verbs as sentence openers, not counting the imperative sentences, and a slight difference (1%) in the number of nouns and adjectives reported.

6.3. Word Usage

The accuracy of identifying word types reflects that of PARTS, which is about 95% correct. The largest source of confusion is between nouns and adjectives. The verb counts were checked on about 20 sentences from each document and found to be about 98% correct.

7. Technical Details

7.1. Finding Sentences

The formatting commands embedded in the text increase the difficulty of finding sentences. Not all text in a document is in sentence form; there are headings, tables, equations and lists, for example. Headings like "Finding Sentences" above should be discarded, not attached to the next sentence. However, since many of the documents are formatted to be phototypeset, and contain font changes, which usually operate on the most important words in the document, discarding all formatting commands is not correct. To improve the programs' ability to find sentence boundaries, the deformatting program, DEROFF [13], has been given some knowledge of the formatting packages used on the UNIX operating system. DEROFF will now do the following:

1. Suppress all formatting macros that are used for titles, headings, author's name, etc.

2. Suppress the arguments to the macros for titles, headings, author's name, etc.
3. Suppress displays, tables, footnotes and text that is centered or in no-fill mode.
4. Substitute a place holder for equations and check for hidden end markers. The place holder is necessary because many typists and authors use the equation setter to change fonts on important words. For this reason, header files containing the definition of the EQN delimiters must also be included as input to STYLE. End markers are often hidden when an equation ends a sentence and the period is typed inside the EQN delimiters.
5. Add a "." after lists. If the flag −ml is also used, all lists are suppressed. This is a separate flag because of the variety of ways the list macros are used. Often, lists are sentences that should be included in the analysis. The user must determine how lists are used in the document to be analyzed.

Both STYLE and DICTION call DEROFF before they look at the text. The user should supply the −ml flag if the document contains many lists of non-sentences that should be skipped.

7.2. Details of DICTION

The program DICTION is based on the string matching program FGREP. FGREP takes as input a file of patterns to be matched and a file to be searched and outputs each line that contains any of the patterns with no indication of which pattern was matched. The following changes have been added to FGREP:

1. The basic unit that DICTION operates on is a sentence rather than a line. Each sentence that contains one of the patterns is output.
2. Upper case letters are mapped to lower case.
3. Punctuation is replaced by blanks.
4 All pattern matches in the sentence are found and surrounded with ``['' '']'' .
5. A method for suppressing a string match has been added. Any pattern that begins with ``~'' will not be matched. Because the matching algorithm finds the longest substring, the suppression of a match allows words in some correct contexts not to be matched while allowing the word in another context to be found. For example, the word ``which'' is often incorrectly used instead of ``that'' in restrictive clauses. However, ``which'' is usually correct when preceded by a preposition or ``,''. The default pattern file suppresses the match of the common prepositions or a double blank followed by ``which'' and therefore matches only the suspect uses. The double blank accounts for the replaced comma.

8. Conclusions

A system of writing tools that measure some of the objective characteristics of writing style has been developed. The tools are sufficiently general that they may be applied to documents on any subject with equal accuracy. Although the measurements are only of the surface structure of the text, they do point out problem areas. In addition to helping writers produce better documents, these programs may be useful for studying the writing process and finding other formulae for measuring readability.

References

1. L. L. Cherry, "PARTS - A System for Assigning Word Classes to English Text," submitted *Communications of the ACM.*
2. B. W. Kernighan and J. R. Mashey, "The UNIX Programming Environment," *Software – Practice & Experience* , **9**, 1-15 (1979).
3. G. R. Klare, "Assessing Readability," *Reading Research Quarterly,* 1974-1975, **10** , 62-102.
4. E. A. Smith and P. Kincaid, "Derivation and validation of the automated readability index for use with technical materials," *Human Factors,* 1970, 12, 457-464.
5. J. P. Kincaid, R. P. Fishburne, R. L. Rogers, and B. S. Chissom, "Derivation of new readability formulas (Automated Readability Index, Fog count, and Flesch Reading Ease Formula) for Navy enlisted personnel," Navy Training Command Research Branch Report 8-75, Feb., 1975.
6. M. Coleman and T. L. Liau, "A Computer Readability Formula Designed for Machine Scoring," *Journal of Applied Psychology,* 1975, 60, 283-284.
7. R. Flesch, "A New Readability Yardstick," *Journal of Applied Psychology,* 1948, 32, 221-233.
8. E. U. Coke, private communication.
9. D. W. Ewing, *Writing for Results,* John Wiley & Sons, Inc., New York, N. Y. (1974).
10. G. Leggett, C. D. Mead and W. Charvat, *Prentice-Hall Handbook for Writers,* Seventh Edition, Prentice-Hall Inc., Englewood Cliffs, N. J. (1978).
11. E. B. Coleman, "Learning of Prose Written in Four Grammatical Transformations," *Journal of Applied Psychology,* 1965, vol. 49, no. 5, pp. 332-341.

12 A. V. Aho and M. J. Corasick, "Efficient String Matching: an aid to Bibliographic Search," *Communications of the ACM,* **18**, (6), 333-340, June 1975.

13. Bell Laboratories, *"UNIX TIME-SHARING SYSTEM: UNIX PROGRAMMER'S MANUAL,"* Seventh Edition, Vol. 1 (January 1979).

Appendix 1

STYLE Abbreviations

a. d.
A. M.
a. m.
b. c.
Ch.
ch.
ckts.
dB.
Dept.
dept.
Depts.
depts.
Dr.
Drs.
e. g.
Eq.
eq.
et al.
etc.
Fig.
fig.
Figs.
figs.
ft.
i. e.
in.
Inc.
Jr.
jr.
mi.
Mr.
Mrs.
Ms.
No.
no.
Nos.
nos.
P. M.
p. m.
Ph. D.
Ph. d.
Ref.
ref.
Refs.
refs.
St.
vs.
yr.

Appendix 2

Default DICTION Patterns

a great deal of
a large number of
a lot of
a majority of
a need for
a number of
a particular preference for
a preference for
a small number of
a tendency to
abovementioned
absolutely complete
absolutely essential
accomplished
accordingly
activate
actual
added increments
adequate enough
advent
afford an opportunity
aggregate
all of
all throughout
along the line
an indication of
analyzation
and etc
and or
another additional
any and all
arrive at a
as a matter of fact
as a method of
as good or better than
as of now
as per
as regards
as related to
as to
assistance
assistance to
assistance to
assuming that
at a later date
at about
at above
at all times
at an early date
at below
at the present
at the time when
at this point in time
at this time
at which time
at your earliest convenience
authorization
awful
basic fundamentals
basically
be cognizant of
being as
being that
brief in duration
bring to a conclusion
but that
but what
by means of
by the use of
carry out experiments
center about
center around
center portion
check into
check on
check up on
circle around
close proximity
collaborate together
collect together
combine together
come to an end
commence
common accord
compensation
completely eliminated
comprise
concerning
conduct an investigation of
conjecture
connect up
consensus of opinion
consequent result
consolidate together
construct
contemplate
continue on
continue to remain
could of
count up
couple together
debate about
decide on
deleterious effect
demean
demonstrate
depreciate in value
deserving of
desirable benefits
desirous of
different than
discontinue
disutility
divide up
doubt but
due to
duly noted
during the time that
each and every
early beginnings
effectuate
emotional feelings
empty out
enclosed herein
enclosed herewith
end result
end up
endeavor
enter in
enter into
enthused
entirely complete
equally good as
essentially
eventuate
every now and then
exactly identical
experiencing difficulty
fabricate
face up to
facilitate
facts and figures
fast in action
fearful of
fearful that
few in number
file away
final completion
final ending
final outcome
final result
finalize
find it interesting to know
first and foremost
first beginnings
first initiated
firstly
follow after
following after
for the purpose of
for the reason that
for the simple reason that
for this reason
for your information
from the point of view of
full and complete
generally agreed
good and
got to
gratuitous
greatly minimize
head up
help but
helps in the production of
hopeful
if and when
if at all possible
impact
implement
important essentials
importantly
in a large measure
in a position to
in accordance
in advance of
in agreement with
in all cases
in back of
in behalf of
in behind
in between
in case
in close proximity
in conflict with
in conjunction with
in connection with
in fact
in large measure
in many cases
in most cases
in my opinion I think
in order to
in rare cases
in reference to
in regard to
in regards to
in relation with
in short supply
in size
in terms of
in the amount of
in the case of
in the course of
in the event
in the field of
in the form of
in the instance of
in the interim
in the last analysis
in the matter of
in the near future
in the neighborhood of
in the not too distant future
in the proximity of
in the range of
in the same way as described
in the shape of
in the vicinity of
in this case
in view of the
in violation of
inasmuch as
indicate
indicative of
initialize
initiate
injurious to
inquire
inside of
institute a
intents and purposes
intermingle
irregardless
is defined as
is used to control
is when
is where
it is incumbent
it stands to reason
it was noted that if
joint cooperation
joint partnership
just exactly
kind of
know about
last but not least
later on
leaving out of consideration
liable
link up
literally
little doubt that
lose out on
lots of
main essentials
make a
make adjustments to
make an
make application to
make contact with
make mention of
make out a list of
make the acquaintance of
make the adjustment
manner
maximum possible
meaningful
meet up with
melt down
melt up
methodology
might of
minimize as far as possible
minor importance
miss out on
modification

more preferable
most unique
must of
mutual cooperation
necessary requisite
necessitate
need for
nice
not be un
not in a position to
not of a high order of accuracy
not un
notwithstanding
of considerable magnitude
of that
of the opinion that
off of
on a few occasions
on account of
on behalf of
on the grounds that
on the occasion
on the part of
one of the
open up
operates to correct
outside of
over with
overall
past history
perceptive of
perform a measurement
perform the measurement
permits the reduction of
personalize
pertaining to
physical size
plan ahead
plan for the future
plan in advance
plan on
present a conclusion
present a report
presently
prior to
prioritize
proceed to
procure
productive of
prolong the duration
protrude out from
provided that
pursuant to
put to use in
range all the way from
reason is because
reason why
recur again
reduce down
refer back
reference to this
reflective of
regarding
regretful
reinitiate
relative to
repeat again
representative of
resultant effect
resume again
retreat back
return again
return back
revert back
seal off
seems apparent
send a communication
short space of time
should of
single unit
situation
so as to
sort of
spell out
still continue
still remain
subsequent
substantially in agreement
succeed in
suggestive of
superior than
surrounding circumstances
take appropriate
take cognizance of
take into consideration
termed as
terminate
termination
the author
the authors
the case that
the fact
the foregoing
the foreseeable future
the fullest possible extent
the majority of
the nature
the necessity of
the only difference being that
the order of
the point that
the truth is
there are not many
through the medium of
through the use of
throughout the entire
time interval
to summarize the above
total effect of all this
totality
transpire
true facts
try and
ultimate end
under a separate cover
under date of
under separate cover
under the necessity to
underlying purpose
undertake a study
uniformly consistent
unique
until such time as
up to this time
upshot
utilize
very
very complete
very unique
vital
which
with a view to
with reference to
with regard to
with the exception of
with the object of
with the result that
with this in mind, it is clear that
within the realm of possibility
without further delay
worth while
would of
ing behavior
wise
¯ which
¯ about which
¯ after which
¯ at which
¯ between which
¯ by which
¯ for which
¯ from which
¯ in which
¯ into which
¯ of which
¯ on which
¯ on which
¯ over which
¯ through which
¯ to which
¯ under which
¯ upon which
¯ with which
¯ without which
¯clockwise
¯likewise
¯otherwise

NROFF/TROFF User's Manual

Joseph F. Ossanna

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

NROFF and TROFF are text processors under the PDP-11 UNIX Time-Sharing System[1] that format text for typewriter-like terminals and for a Graphic Systems phototypesetter, respectively. They accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. NROFF and TROFF offer unusual freedom in document styling, including: arbitrary style headers and footers; arbitrary style footnotes; multiple automatic sequence numbering for paragraphs, sections, etc; multiple column output; dynamic font and point-size control; arbitrary horizontal and vertical local motions at any point; and a family of automatic overstriking, bracket construction, and line drawing functions.

NROFF and TROFF are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program. NROFF can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

Usage

The general form of invoking NROFF (or TROFF) at UNIX command level is

nroff *options files* (or **troff** *options files*)

where *options* represents any of a number of option arguments and *files* represents the list of files containing the document to be formatted. An argument consisting of a single minus (**−**) is taken to be a file name corresponding to the standard input. If no file names are given input is taken from the standard input. The options, which may appear in any order so long as they appear before the files, are:

Option	*Effect*
−o*list*	Print only pages whose page numbers appear in *list*, which consists of comma-separated numbers and number ranges. A number range has the form *N*−*M* and means pages *N* through *M;* a initial −*N* means from the beginning to page *N;* and a final *N*− means from *N* to the end.
−n*N*	Number first generated page *N*.
−s*N*	Stop every *N* pages. NROFF will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a newline. TROFF will stop the phototypesetter every *N* pages, produce a trailer to allow changing cassettes, and will resume after the phototypesetter START button is pressed.
−m*name*	Prepends the macro file **/usr/lib/tmac.***name* to the input *files.*
−r*aN*	Register *a* (one-character) is set to *N*.
−i	Read standard input after the input files are exhausted.
−q	Invoke the simultaneous input-output mode of the **rd** request.

NROFF Only

−T*name*	Specifies the name of the output terminal type. Currently defined names are **37** for the (default) Model 37 Teletype®, **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300S** for the DASI-300S, **300** for the DASI-300, and **450** for the DASI-450 (Diablo Hyterm).
−e	Produce equally-spaced words in adjusted lines, using full terminal resolution.

TROFF Only

−t	Direct output to the standard output instead of the phototypesetter.
−f	Refrain from feeding out paper and stopping phototypesetter at the end of the run.
−w	Wait until phototypesetter is available, if currently busy.
−b	TROFF will report whether the phototypesetter is busy or available. No text processing is done.
−a	Send a printable (ASCII) approximation of the results to the standard output.
−p*N*	Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.
−g	Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output.

Each option is invoked as a separate argument; for example,

nroff −o*4,8−10* **−T***300S* **−m***abc file1 file2*

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named *file1* and *file2*, specifies the output terminal as a DASI-300S, and invokes the macro package *abc*.

Various pre- and post-processors are available for use with NROFF and TROFF. These include the equation preprocessors NEQN and EQN[2] (for NROFF and TROFF respectively), and the table-construction preprocessor TBL[3]. A reverse-line postprocessor COL[4] is available for multiple-column NROFF output on terminals without reverse-line ability; COL expects the Model 37 Teletype escape sequences that NROFF produces by default. TK[4] is a 37 Teletype simulator postprocessor for printing NROFF output on a Tektronix 4014. TCAT[4] is phototypesetter-simulator postprocessor for TROFF that produces an approximation of phototypesetter output on a Tektronix 4014. For example, in

tbl *files* | **eqn** | **troff −t** *options* | **tcat**

the first | indicates the piping of TBL's output to EQN's input; the second the piping of EQN's output to TROFF's input; and the third indicates the piping of TROFF's output to TCAT. GCAT[4] can be used to send TROFF (**−g**) output to the Murray Hill Computation Center.

The remainder of this manual consists of: a Summary and Index; a Reference Manual keyed to the index; and a set of Tutorial Examples. Another tutorial is [5].

Joseph F. Ossanna

References

[1] K. Thompson, D. M. Ritchie, *UNIX Programmer's Manual*, Sixth Edition (May 1975).

[2] B. W. Kernighan, L. L. Cherry, *Typesetting Mathematics — User's Guide (Second Edition)*, Bell Laboratories internal memorandum.

[3] M. E. Lesk, *Tbl — A Program to Format Tables*, Bell Laboratories internal memorandum.

[4] Internal on-line documentation, on UNIX.

[5] B. W. Kernighan, *A TROFF Tutorial*, Bell Laboratories internal memorandum.

SUMMARY AND INDEX

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*
1. General Explanation				
2. Font and Character Size Control				
.ps ±*N*	10 point	previous	E	Point size; also \s±*N*.†
.ss *N*	12/36 em	ignored	E	Space-character size set to *N*/36 em.†
.cs *FNM*	off	-	P	Constant character space (width) mode (font *F*).†
.bd *F N*	off	-	P	Embolden font *F* by *N*−1 units.†
.bd S *F N*	off	-	P	Embolden Special Font when current font is *F*.†
.ft *F*	Roman	previous	E	Change to font *F* = *x*, *xx*, or 1-4. Also \f*x*, \f(*xx*, \f*N*.
.fp *N F*	R,I,B,S	ignored	-	Font named *F* mounted on physical position 1≤*N*≤4.
3. Page Control				
.pl ±*N*	11 in	11 in	v	Page length.
.bp ±*N*	*N*=1	-	B‡,v	Eject current page; next page number *N*.
.pn ±*N*	*N*=1	ignored	-	Next page number *N*.
.po ±*N*	0; 26/27 in	previous	v	Page offset.
.ne *N*	-	*N*=1 *V*	D,v	Need *N* vertical space (*V* = vertical spacing).
.mk *R*	none	internal	D	Mark current vertical place in register *R*.
.rt ±*N*	none	internal	D,v	Return *(upward only)* to marked vertical place.
4. Text Filling, Adjusting, and Centering				
.br	-	-	B	Break.
.fi	fill	-	B,E	Fill output lines.
.nf	fill	-	B,E	No filling or adjusting of output lines.
.ad *c*	adj,both	adjust	E	Adjust output lines with mode *c*.
.na	adjust	-	E	No output line adjusting.
.ce *N*	off	*N*=1	B,E	Center following *N* input text lines.
5. Vertical Spacing				
.vs *N*	1/6in;12pts	previous	E,p	Vertical base line spacing (*V*).
.ls *N*	*N*=1	previous	E	Output *N*−1 *V*s after each text output line.
.sp *N*	-	*N*=1 *V*	B,v	Space vertical distance *N in either direction.*
.sv *N*	-	*N*=1 *V*	v	Save vertical distance *N*.
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.
6. Line Length and Indenting				
.ll ±*N*	6.5 in	previous	E,m	Line length.
.in ±*N*	*N*=0	previous	B,E,m	Indent.
.ti ±*N*	-	ignored	B,E,m	Temporary indent.
7. Macros, Strings, Diversion, and Position Traps				
.de *xx yy*	-	*.yy*=..	-	Define or redefine macro *xx;* end at call of *yy*.
.am *xx yy*	-	*.yy*=..	-	Append to a macro.
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*.
.as *xx string*	-	ignored	-	Append *string* to string *xx*.

*Values separated by ";" are for NROFF and TROFF respectively.

#Notes are explained at the end of this Summary and Index

†No effect in NROFF.

‡The use of " ' " as control character (instead of ".") suppresses the break function.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.rm *xx*	-	ignored	-	Remove request, macro, or string.
.rn *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*.
.di *xx*	-	end	D	Divert output to macro *xx*.
.da *xx*	-	end	D	Divert and append to *xx*.
.wh *N xx*	-	-	v	Set location trap; negative is w.r.t. page bottom.
.ch *xx N*	-	-	v	Change trap location.
.dt *N xx*	-	off	D,v	Set a diversion trap.
.it *N xx*	-	off	E	Set an input-line count trap.
.em *xx*	none	none	-	End macro is *xx*.

8. Number Registers

.nr *R* ±*N M*		-	u	Define and set number register *R*; auto-increment by *M*.
.af *R c*	arabic	-	-	Assign format to register *R* (*c*=1, i, I, a, A).
.rr *R*	-	-	-	Remove register *R*.

9. Tabs, Leaders, and Fields

.ta *Nt* ...	0.8; 0.5in	none	E,m	Tab settings; *left* type, unless *t*=R(right), C(centered).
.tc *c*	none	none	E	Tab repetition character.
.lc *c*	.	none	E	Leader repetition character.
.fc *a b*	off	off	-	Set field delimiter *a* and pad character *b*.

10. Input and Output Conventions and Character Translations

.ec *c*	\	\	-	Set escape character.
.eo	on	-	-	Turn off escape character mechanism.
.lg *N*	-; on	on	-	Ligature mode on if *N*>0.
.ul *N*	off	*N*=1	E	Underline (italicize in TROFF) *N* input lines.
.cu *N*	off	*N*=1	E	Continuous underline in NROFF; like **ul** in TROFF.
.uf *F*	Italic	Italic	-	Underline font set to *F* (to be switched to by **ul**).
.cc *c*	.	.	E	Set control character to *c*.
.c2 *c*	'	'	E	Set nobreak control character to *c*.
.tr *abcd....*	none	-	O	Translate *a* to *b*, etc. on output.

11. Local Horizontal and Vertical Motions, and the Width Function

12. Overstrike, Bracket, Line-drawing, and Zero-width Functions

13. Hyphenation.

.nh	hyphenate	-	E	No hyphenation.
.hy *N*	hyphenate	hyphenate	E	Hyphenate; *N* = mode.
.hc *c*	\%	\%	E	Hyphenation indicator character *c*.
.hw *word1* ...		ignored	-	Exception words.

14. Three Part Titles.

.tl *'left'center'right'*		-	-	Three part title.
.pc *c*	%	off	-	Page number character.
.lt ±*N*	6.5 in	previous	E,m	Length of title.

15. Output Line Numbering.

.nm ±*N M S I*		off	E	Number mode on or off, set parameters.
.nn *N*	-	*N*=1	E	Do not number next *N* lines.

16. Conditional Acceptance of Input

.if *c anything*		-	-	If condition *c* true, accept *anything* as input, for multi-line use \\{*anything*\\}.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.if !*c anything*		-	-	If condition *c* false, accept *anything*.
.if *N anything*		-	u	If expression $N > 0$, accept *anything*.
.if !*N anything*		-	u	If expression $N \leqslant 0$, accept *anything*.
.if *'string1'string2' anything*			-	If *string1* identical to *string2*, accept *anything*.
.if !*'string1'string2' anything*			-	If *string1* not identical to *string2*, accept *anything*.
.ie *c anything*		-	u	If portion of if-else; all above forms (like if).
.el *anything*		-	-	Else portion of if-else.
17. Environment Switching.				
.ev *N*	*N*=0	previous	-	Environment switched (*push down*).
18. Insertions from the Standard Input				
.rd *prompt*	-	*prompt*=BEL	-	Read insertion.
.ex	-	-	-	Exit from NROFF/TROFF.
19. Input/Output File Switching				
.so *filename*		-	-	Switch source file (*push down*).
.nx *filename*		end-of-file	-	Next file.
.pi *program*		-	-	Pipe output to *program* (NROFF only).
20. Miscellaneous				
.mc *c N*	-	off	E,m	Set margin character *c* and separation *N*.
.tm *string*	-	newline	-	Print *string* on terminal (UNIX standard message output).
.ig *yy*	-	*.yy*=..	-	Ignore till call of *yy*.
.pm *t*	-	all	-	Print macro names and sizes; if *t* present, print only total of sizes.
.fl	-	-	B	Flush output buffer.
21. Output and Error Messages				

Notes-

B	Request normally causes a break.
D	Mode or relevant parameters associated with current diversion level.
E	Relevant parameters are a part of the current environment.
O	Must stay in effect until logical output.
P	Mode must be still or again in effect at the time of physical output.
v,p,m,u	Default scale indicator; if not specified, scale indicators are *ignored*.

Alphabetical Request and Section Number Cross Reference

ad 4	cc 10	ds 7	fc 9	ie 16	ll 6	nh 13	pi 19	rn 7	ta 9	vs 5
af 8	ce 4	dt 7	fi 4	if 16	ls 5	nm 15	pl 3	rr 8	tc 9	wh 7
am 7	ch 7	ec 10	fl 20	ig 20	lt 14	nn 15	pm 20	rs 5	ti 6	
as 7	cs 2	el 16	fp 2	in 6	mc 20	nr 8	pn 3	rt 3	tl 14	
bd 2	cu 10	em 7	ft 2	it 7	mk 3	ns 5	po 3	so 19	tm 20	
bp 3	da 7	eo 10	hc 13	lc 9	na 4	nx 19	ps 2	sp 5	tr 10	
br 4	de 7	ev 17	hw 13	lg 10	ne 3	os 5	rd 18	ss 2	uf 10	
c2 10	di 7	ex 18	hy 13	li 10	nf 4	pc 14	rm 7	sv 5	ul 10	

Escape Sequences for Characters, Indicators, and Functions

Section Reference	*Escape Sequence*	*Meaning*
10.1	\\	\ (to prevent or delay the interpretation of \ )
10.1	\e	Printable version of the *current* escape character.
2.1	\'	' (acute accent); equivalent to \(aa
2.1	\`	` (grave accent); equivalent to \(ga
2.1	\-	− Minus sign in the *current* font
7	\.	Period (dot) (see **de**)
11.1	\(space)	Unpaddable space-size space character
11.1	\0	Digit width space
11.1	\\|	1/6 em narrow space character (zero width in NROFF)
11.1	\^	1/12 em half-narrow space character (zero width in NROFF)
4.1	\&	Non-printing, zero width character
10.6	\!	Transparent line indicator
10.7	\"	Beginning of comment
7.3	\$*N*	Interpolate argument $1 \leqslant N \leqslant 9$
13	\%	Default optional hyphenation character
2.1	\(*xx*	Character named *xx*
7.1	***x*, *(*xx*	Interpolate string *x* or *xx*
9.1	\a	Non-interpreted leader character
12.3	\b'*abc...*'	Bracket building function
4.2	\c	Interrupt text processing
11.1	\d	Forward (down) 1/2 em vertical motion (1/2 line in NROFF)
2.2	\f*x*,\f(*xx*,\f*N*	Change to font named *x* or *xx*, or position *N*
11.1	\h'*N*'	Local horizontal motion; move right *N* *(negative left)*
11.3	\k*x*	Mark horizontal *input* place in register *x*
12.4	\l'*Nc*'	Horizontal line drawing function (optionally with *c*)
12.4	\L'*Nc*'	Vertical line drawing function (optionally with *c*)
8	\n*x*,\n(*xx*	Interpolate number register *x* or *xx*
12.1	\o'*abc...*'	Overstrike characters *a, b, c, ...*
4.1	\p	Break and spread output line
11.1	\r	Reverse 1 em vertical motion (reverse line in NROFF)
2.3	\s*N*, \s±*N*	Point-size change function
9.1	\t	Non-interpreted horizontal tab
11.1	\u	Reverse (up) 1/2 em vertical motion (1/2 line in NROFF)
11.1	\v'*N*'	Local vertical motion; move down *N* *(negative up)*
11.2	\w'*string*'	Interpolate width of *string*
5.2	\x'*N*'	Extra line-space function *(negative before, positive after)*
12.2	\z*c*	Print *c* with zero width (without spacing)
16	\{	Begin conditional input
16	\}	End conditional input
10.7	\(newline)	Concealed (ignored) newline
-	\X	*X*, any character *not* listed above

The escape sequences \\, \., \", \$, *, \a, \n, \t, and \(newline) are interpreted in *copy mode* (§7.2).

Predefined General Number Registers

Section Reference	*Register Name*	*Description*
3	**%**	Current page number.
11.2	**ct**	Character type (set by *width* function).
7.4	**dl**	Width (maximum) of last completed diversion.
7.4	**dn**	Height (vertical size) of last completed diversion.
-	**dw**	Current day of the week (1-7).
-	**dy**	Current day of the month (1-31).
11.3	**hp**	Current horizontal place on *input* line.
15	**ln**	Output line number.
-	**mo**	Current month (1-12).
4.1	**nl**	Vertical position of last printed text base-line.
11.2	**sb**	Depth of string below base line (generated by *width* function).
11.2	**st**	Height of string above base line (generated by *width* function).
-	**yr**	Last two digits of current year.

Predefined Read-Only Number Registers

Section Reference	*Register Name*	*Description*
7.3	**.$**	Number of arguments available at the current macro level.
-	**.A**	Set to 1 in TROFF, if **−a** option used; always 1 in NROFF.
11.1	**.H**	Available horizontal resolution in basic units.
-	**.T**	Set to 1 in NROFF, if **−T** option used; always 0 in TROFF.
11.1	**.V**	Available vertical resolution in basic units.
5.2	**.a**	Post-line extra line-space most recently utilized using **\x**'*N*'.
-	**.c**	Number of *lines* read from current input file.
7.4	**.d**	Current vertical place in current diversion; equal to **nl**, if no diversion.
2.2	**.f**	Current font as physical quadrant (1-4).
4	**.h**	Text base-line high-water mark on current page or diversion.
6	**.i**	Current indent.
6	**.l**	Current line length.
4	**.n**	Length of text portion on previous output line.
3	**.o**	Current page offset.
3	**.p**	Current page length.
2.3	**.s**	Current point size.
7.5	**.t**	Distance to the next trap.
4.1	**.u**	Equal to 1 in fill mode and 0 in nofill mode.
5.1	**.v**	Current vertical line spacing.
11.2	**.w**	Width of previous character.
-	**.x**	Reserved version-dependent register.
-	**.y**	Reserved version-dependent register.
7.4	**.z**	Name of current diversion.

REFERENCE MANUAL

1. General Explanation

1.1. Form of input. Input consists of *text lines*, which are destined to be printed, interspersed with *control lines*, which set parameters or otherwise control subsequent processing. Control lines begin with a *control character*—normally . (period) or ' (acute accent)—followed by a one or two character name that specifies a basic *request* or the substitution of a user-defined *macro* in place of the control line. The control character ' suppresses the *break* function—the forced output of a partially filled line—caused by certain requests. The control character may be separated from the request/macro name by white space (spaces and/or tabs) for esthetic reasons. Names must be followed by either space or newline. Control lines with unrecognized names are ignored.

Various special functions may be introduced anywhere in the input by means of an *escape* character, normally \. For example, the function **\n***R* causes the interpolation of the contents of the *number register R* in place of the function; here *R* is either a single character name as in **\n***x*, or left-parenthesis-introduced, two-character name as in **\n(***xx*.

1.2. Formatter and device resolution. TROFF internally uses 432 units/inch, corresponding to the Graphic Systems phototypesetter which has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. NROFF internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various typewriter-like output devices. TROFF rounds horizontal/vertical numerical parameter input to the actual horizontal/vertical resolution of the Graphic Systems typesetter. NROFF similarly rounds numerical input to the actual resolution of the output device indicated by the **−T** option (default Model 37 Teletype).

1.3. Numerical parameter input. Both NROFF and TROFF accept numerical input with the appended scale indicators shown in the following table, where *S* is the current type size in points, *V* is the current vertical line spacing in basic units, and *C* is a *nominal character width* in basic units.

Scale Indicator	Meaning	Number of basic units TROFF	NROFF
i	Inch	432	240
c	Centimeter	432×50/127	240×50/127
P	Pica = 1/6 inch	72	240/6
m	Em = *S* points	6×*S*	*C*
n	En = Em/2	3×*S*	*C, same as Em*
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	*V*	*V*
none	Default, see below		

In NROFF, *both* the em and the en are taken to be equal to the *C*, which is output-device dependent; common values are 1/10 and 1/12 inch. Actual character widths in NROFF need not be all the same and constructed characters such as −> (→) are often extra wide. The default scaling is ems for the horizontally-oriented requests and functions **ll**, **in**, **ti**, **ta**, **lt**, **po**, **mc**, **\h**, and **\l**; *V*s for the vertically-oriented requests and functions **pl**, **wh**, **ch**, **dt**, **sp**, **sv**, **ne**, **rt**, **\v**, **\x**, and **\L**; **p** for the **vs** request; and **u** for the requests **nr**, **if**, and **ie**. *All* other requests ignore any scale indicators. When a number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator **u** may need to be appended to prevent an additional inappropriate default scaling.

The number, *N*, may be specified in decimal-fraction form but the parameter finally stored is rounded to an integer number of basic units.

The *absolute position* indicator | may be prepended to a number *N* to generate the distance to the vertical or horizontal place *N*. For vertically-oriented requests and functions, |*N* becomes the distance in basic units from the current vertical place on the page or in a *diversion* (§7.4) to the the vertical place *N*. For *all* other requests and functions, |*N* becomes the distance from the current horizontal place on the *input* line to the horizontal place *N*. For example,

```
.sp |3.2c
```

will space *in the required direction* to 3.2 centimeters from the top of the page.

1.4. Numerical expressions. Wherever numerical input is expected an expression involving parentheses, the arithmetic operators +, −, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), : (or) may be used. Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or − is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to *every* number in an expression for which the desired and default scaling differ. For example, if the number register **x** contains 2 and the current point size is 10, then

```
.ll (4.25i+\nxP+3)/2u
```

will set the line length to 1/2 the sum of 4.25 inches + 2 picas + 30 points.

1.5. Notation. Numerical parameters are indicated in this manual in two ways. ±*N* means that the argument may take the forms *N*, +*N*, or −*N* and that the corresponding effect is to set the affected parameter to *N*, to increment it by *N*, or to decrement it by *N* respectively. Plain *N* means that an initial algebraic sign is *not* an increment indicator, but merely the sign of *N*. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are **sp**, **wh**, **ch**, **nr**, and **if**. The requests **ps**, **ft**, **po**, **vs**, **ls**, **ll**, **in**, and **lt** restore the *previous* parameter value in the *absence* of an argument.

Single character arguments are indicated by single lower case letters and one/two character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

2. Font and Character Size Control

2.1. Character set. The TROFF character set consists of the Graphics Systems Commercial II character set plus a Special Mathematical Font character set—each having 102 characters. These character sets are shown in the attached Table I. All ASCII characters are included, with some on the Special Font. With three exceptions, the ASCII characters are input as themselves, and non-ASCII characters are input in the form \\(*xx* where *xx* is a two-character name given in the attached Table II. The three ASCII exceptions are mapped as follows:

ASCII Input		Printed by TROFF	
Character	Name	Character	Name
´	acute accent	’	close quote
\`	grave accent	‘	open quote
−	minus	-	hyphen

The characters ´, \`, and − may be input by \\´, \\\`, and \\− respectively or by their names (Table II). The ASCII characters @, #, ", ´, \`, <, >, \\, {, }, ~, ^, and _ exist only on the Special Font and are printed as a 1-em space if that Font is not mounted.

NROFF understands the entire TROFF character set, but can in general print only ASCII characters, additional characters as may be available on the output device, such characters as may be able to be constructed by overstriking or other combination, and those that can reasonably be mapped into other printable characters. The exact behavior is determined by a driving table prepared for each device. The characters ´, `, and _ print as themselves.

2.2. Fonts. The default mounted fonts are Times Roman (R), Times Italic (I), Times Bold (B), and the Special Mathematical Font (S) on physical typesetter positions 1, 2, 3, and 4 respectively. These fonts are used in this document. The *current* font, initially Roman, may be changed (among the mounted fonts) by use of the **ft** request, or by imbedding at any desired point either \f*x*, \f(*xx*, or \f*N* where *x* and *xx* are the name of a mounted font and *N* is a numerical font position. It is *not* necessary to change to the Special font; characters on that font are automatically handled. A request for a named but not-mounted font is *ignored*. TROFF can be informed that any particular font is mounted by use of the **fp** request. The list of known fonts is installation dependent. In the subsequent discussion of font-related requests, *F* represents either a one/two-character font name or the numerical font position, 1-4. The current font is available (as numerical position) in the read-only number register **.f**.

NROFF understands font control and normally underlines Italic characters (see §10.5).

2.3. Character size. Character point sizes available on the Graphic Systems typesetter are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The **ps** request is used to change or restore the point size. Alternatively the point size may be changed between any two characters by imbedding a \s*N* at the desired point to set the size to *N*, or a \s±*N* (1⩽*N*⩽9) to increment/decrement the size by *N*; \s0 restores the *previous* size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the **.s** register. NROFF ignores type size control.

Request Form	***Initial Value***	***If No Argument***	***Notes****	***Explanation***
.ps ±*N*	10 point	previous	E	Point size set to ±*N*. Alternatively imbed \s*N* or \s±*N*. Any positive size value may be requested; if invalid, the next larger valid size will result, with a maximum of 36. A paired sequence +*N*, −*N* will work because the previous requested value is also remembered. Ignored in NROFF.
.ss *N*	12/36 em	ignored	E	Space-character size is set to *N*/36 ems. This size is the minimum word spacing in adjusted text. Ignored in NROFF.
.cs *FNM*	off	-	P	Constant character space (width) mode is set on for font *F* (if mounted); the width of every character will be taken to be *N*/36 ems. If *M* is absent, the em is that of the character's point size; if *M* is given, the em is *M*-points. All affected characters are centered in this space, including those with an actual width larger than this space. Special Font characters occurring while the current font is *F* are also so treated. If *N* is absent, the mode is turned off. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.
.bd *F N*	off	-	P	The characters in font *F* will be artificially emboldened by printing each one twice, separated by *N*−1 basic units. A reasonable value for *N* is 3 when the character size is in the vicinity of 10 points. If *N* is missing the embolden mode is turned off. The column heads above were printed with **.bd** I 3. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.

*Notes are explained at the end of the Summary and Index above.

.bd S *F N*	off	-	P	The characters in the Special Font will be emboldened whenever the current font is *F*. This manual was printed with **.bd S B 3**. The mode must be still or again in effect when the characters are physically printed.
.ft *F*	Roman	previous	E	Font changed to *F*. Alternatively, imbed **\f***F*. The font name **P** is reserved to mean the previous font.
.fp *N F*	R,I,B,S	ignored	-	Font position. This is a statement that a font named *F* is mounted on position *N* (1-4). It is a fatal error if *F* is not known. The phototypesetter has four fonts physically mounted. Each font consists of a film strip which can be mounted on a numbered quadrant of a wheel. The default mounting sequence assumed by TROFF is R, I, B, and S on positions 1, 2, 3 and 4.

3. Page control

Top and bottom margins are *not* automatically provided; it is conventional to define two *macros* and to set *traps* for them at vertical positions 0 (top) and $-N$ (N from the bottom). See §7 and Tutorial Examples §T2. A pseudo-page transition onto the *first* page occurs either when the first *break* occurs or when the first *non-diverted* text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. In the following, references to the *current diversion* (§7.4) mean that the mechanism being described works during both ordinary and diverted output (the former considered as the top diversion level).

The useable page width on the Graphic Systems phototypesetter is about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper. The physical limitations on NROFF output are output-device dependent.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.pl $\pm N$	11 in	11 in	v	Page length set to $\pm N$. The internal limitation is about 75 inches in TROFF and about 136 inches in NROFF. The current page length is available in the **.p** register.
.bp $\pm N$	N=1	-	B*,v	Begin page. The current page is ejected and a new page is begun. If $\pm N$ is given, the new page number will be $\pm N$. Also see request **ns**.
.pn $\pm N$	N=1	ignored	-	Page number. The next page (when it occurs) will have the page number $\pm N$. A **pn** must occur before the initial pseudo-page transition to effect the page number of the first page. The current page number is in the **%** register.
.po $\pm N$	0; 26/27 in†	previous	v	Page offset. The current *left margin* is set to $\pm N$. The TROFF initial value provides about 1 inch of paper margin including the physical typesetter margin of 1/27 inch. In TROFF the maximum (line-length)+(page-offset) is about 7.54 inches. See §6. The current page offset is available in the **.o** register.
.ne *N*	-	N=1 *V*	D,v	Need *N* vertical space. If the distance, *D*, to the next trap position (see §7.5) is less than *N*, a forward vertical space of size *D* occurs, which will spring the trap. If there are no remaining traps on the page, *D* is the

*The use of " ' " as control character (instead of ".") suppresses the break function.

†Values separated by ";" are for NROFF and TROFF respectively.

				distance to the bottom of the page. If $D < V$, another line could still be output and spring the trap. In a diversion, D is the distance to the *diversion trap*, if any, or is very large.
.mk *R*	none	internal	D	Mark the *current* vertical place in an internal register (both associated with the current diversion level), or in register *R*, if given. See **rt** request.
.rt ±*N*	none	internal	D,v	Return *upward only* to a marked vertical place in the current diversion. If ±*N* (w.r.t. current place) is given, the place is ±*N* from the top of the page or diversion or, if *N* is absent, to a place marked by a previous **mk**. Note that the **sp** request (§5.3) may be used in all cases instead of **rt** by spacing to the absolute place stored in a explicit register; e. g. using the sequence **.mk** *R* ... **.sp** |**\n***R***u**.

4. Text Filling, Adjusting, and Centering

4.1. Filling and adjusting. Normally, words are collected from input text lines and assembled into a output text line until some word doesn't fit. An attempt is then made the hyphenate the word in effort to assemble a part of it into the output line. The spaces between the words on the output line are then increased to spread out the line to the current *line length* minus any current *indent*. A *word* is any string of characters delimited by the *space* character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the *unpaddable space* character "\ " (backslash-space). The adjusted word spacings are uniform in TROFF and the minimum interword spacing can be controlled with the **ss** request (§2). In NROFF, they are normally nonuniform because of quantization to character-size spaces; however, the command line option **−e** causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation (§13) can all be prevented or controlled. The *text length* on the last line output is available in the **.n** register, and text base-line position on the page for this line is in the **nl** register. The text base-line high-water mark (lowest place) on the current page is in the **.h** register.

An input text line ending with **.**, **?**, or **!** is taken to be the end of a *sentence*, and an additional space character is automatically provided during filling. Multiple inter-word space characters found in the input are retained, except for trailing spaces; initial spaces also cause a *break*.

When filling is in effect, a **\p** may be imbedded or attached to a word to cause a *break* at the *end* of the word and have the resulting output line *spread out* to fill the current line length.

A text input line that happens to begin with a control character can be made to not look like a control line by prefacing it with the non-printing, zero-width filler character **\&**. Still another way is to specify output translation of some convenient character into the control character using **tr** (§10.5).

4.2. Interrupted text. The copying of a input line in *nofill* (non-fill) mode can be *interrupted* by terminating the partial line with a **\c**. The *next* encountered input text line will be considered to be a continuation of the same line of input text. Similarly, a word within *filled* text may be interrupted by terminating the word (and line) with **\c**; the next encountered text will be taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line will be forced out along with any partial word.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.br	-	-	B	Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.

.fi	fill on	-	B,E	Fill subsequent output lines. The register **.u** is 1 in fill mode and 0 in nofill mode.
.nf	fill on	-	B,E	Nofill. Subsequent output lines are *neither* filled *nor* adjusted. Input text lines are copied directly to output lines *without regard* for the current line length.
.ad *c*	adj,both	adjust	E	Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator *c* is present, the adjustment type is changed as shown in the following table.

Indicator	Adjust Type
l	adjust left margin only
r	adjust right margin only
c	center
b or **n**	adjust both margins
absent	unchanged

.na	adjust	-	E	Noadjust. Adjustment is turned off; the right margin will be ragged. The adjustment type for **ad** is not changed. Output line filling still occurs if fill mode is on.
.ce *N*	off	*N*=1	B,E	Center the next *N* input text lines within the current (line-length minus indent). If *N*=0, any residual count is cleared. A break occurs after each of the *N* input lines. If the input line is too long, it will be left adjusted.

5. Vertical Spacing

5.1. Base-line spacing. The vertical spacing *(V)* between the base-lines of successive output lines can be set using the **vs** request with a resolution of 1/144 inch = 1/2 point in TROFF, and to the output device resolution in NROFF. *V* must be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set *V* to 2 points greater than the point size; TROFF default is 10-point type on a 12-point spacing (as in this document). The current *V* is available in the **.v** register. Multiple-*V* line separation (e.g. double spacing) may be requested with **ls**.

5.2. Extra line-space. If a word contains a vertically tall construct requiring the output line containing it to have extra vertical space before and/or after it, the *extra-line-space* function **\x**'*N*' can be imbedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter (here '), the delimiter choice is arbitrary, except that it can't look like the continuation of a number expression for *N*. If *N* is negative, the output line containing the word will be preceded by *N* extra vertical space; if *N* is positive, the output line containing the word will be followed by *N* extra vertical space. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the **.a** register.

5.3. Blocks of vertical space. A block of vertical space is ordinarily requested using **sp**, which honors the *no-space* mode and which does not space *past* a trap. A contiguous block of vertical space may be reserved using **sv**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.vs *N*	1/6in;12pts	previous	E,p	Set vertical base-line spacing size *V*. Transient *extra* vertical space available with **\x**'*N*' (see above).
.ls *N*	*N*=1	previous	E	*Line* spacing set to ±*N*. *N*−1 *V*s *(blank lines)* are appended to each output text line. Appended blank lines are omitted, if the text or previous appended blank line

				reached a trap position.
.sp *N*	-	*N*=1 *V*	B,v	Space vertically in *either* direction. If *N* is negative, the motion is *backward* (upward) and is limited to the distance to the top of the page. Forward (downward) motion is truncated to the distance to the nearest trap. If the no-space mode is on, no spacing occurs (see **ns**, and **rs** below).
.sv *N*	-	*N*=1 *V*	v	Save a contiguous vertical block of size *N*. If the distance to the next trap is greater than *N*, *N* vertical space is output. No-space mode has *no* effect. If this distance is less than *N*, no vertical space is immediately output, but *N* is remembered for later output (see **os**). Subsequent **sv** requests will overwrite any still remembered *N*.
.os	-	-	-	Output saved vertical space. No-space mode has *no* effect. Used to finally output a block of vertical space requested by an earlier **sv** request.
.ns	space	-	D	No-space mode turned on. When on, the no-space mode inhibits **sp** requests and **bp** requests *without* a next page number. The no-space mode is turned off when a line of output occurs, or with **rs**.
.rs	space	-	D	Restore spacing. The no-space mode is turned off.
Blank text line.		-	B	Causes a break and output of a blank line exactly like **sp 1**.

6. Line Length and Indenting

The maximum line length for fill mode may be set with **ll**. The indent may be set with **in**; an indent applicable to *only* the *next* output line may be set with **ti**. The line length includes indent space but *not* page offset space. The line-length minus the indent is the basis for centering with **ce**. The effect of **ll**, **in**, or **ti** is delayed, if a partially collected line exists, until after that line is output. In fill mode the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers **.l** and **.i** respectively. The length of *three-part titles* produced by **tl** (see §14) is *independently* set by **lt**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ll ±*N*	6.5 in	previous	E,m	Line length is set to ±*N*. In TROFF the maximum (line-length)+(page-offset) is about 7.54 inches.
.in ±*N*	*N*=0	previous	B,E,m	Indent is set to ±*N*. The indent is prepended to each output line.
.ti ±*N*	-	ignored	B,E,m	Temporary indent. The *next* output text line will be indented a distance ±*N* with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.

7. Macros, Strings, Diversion, and Position Traps

7.1. Macros and strings. A *macro* is a named set of arbitrary *lines* that may be invoked by name or with a *trap*. A *string* is a named string of *characters*, *not* including a newline character, that may be interpolated by name at any point. Request, macro, and string names share the *same* name list. Macro and string names may be one or two characters long and may usurp previously defined request, macro, or string names. Any of these entities may be renamed with **rn** or removed with **rm**. Macros are created by **de** and **di**, and appended to by **am** and **da**; **di** and **da** cause normal output to be stored in a macro. Strings are created by **ds** and appended to by **as**. A macro is invoked in the same way as a request; a control line beginning *.xx* will interpolate the contents of macro *xx*. The remainder of the line may contain up to nine *arguments*. The strings *x* and *xx* are interpolated at any desired point with \•*x* and \•(*xx* respectively. String references and macro invocations may be nested.

7.2. Copy mode input interpretation. During the definition and extension of strings and macros (not by diversion) the input is read in *copy mode*. The input is copied without interpretation *except* that:

- The contents of number registers indicated by **\n** are interpolated.
- Strings indicated by **\•** are interpolated.
- Arguments indicated by **\$** are interpolated.
- Concealed newlines indicated by \(newline) are eliminated.
- Comments indicated by **\"** are eliminated.
- **\t** and **\a** are interpreted as ASCII horizontal tab and SOH respectively (§9).
- \\ is interpreted as \.
- \. is interpreted as ".".

These interpretations can be suppressed by prepending a \. For example, since \\ maps into a \, **n** will copy as **\n** which will be interpreted as a number register indicator when the macro or string is reread.

7.3. Arguments. When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character, and arguments may be surrounded by double-quotes to permit imbedded space characters. Pairs of double-quotes may be imbedded in double-quoted arguments to represent a single double-quote. If the desired arguments won't fit on a line, a concealed newline may be used to continue on the next line.

When a macro is invoked the *input level* is *pushed down* and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at *any* point within the macro with **\$***N*, which interpolates the *N*th argument ($1 \leqslant N \leqslant 9$). If an invoked argument doesn't exist, a null string results. For example, the macro *xx* may be defined by

```
.de xx        \"begin definition
Today is \\$1 the \\$2.
..            \"end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

Today is Monday the 14th.

Note that the **\$** was concealed in the definition with a prepended \. The number of currently available arguments is in the **.$** register.

No arguments are available at the top (non-macro) level in this implementation. Because string referencing is implemented as a input-level push down, no arguments are available from *within* a string. No arguments are available within a trap-invoked macro.

Arguments are copied in *copy mode* onto a stack where they are available for reference. The mechanism does not allow an argument to contain a direct reference to a *long* string (interpolated at copy time) and it is advisable to conceal string references (with an extra \) to delay interpolation until argument reference time.

7.4. Diversions. Processed output may be diverted into a macro for purposes such as footnote processing (see Tutorial §T5) or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap may be set at a specified vertical position. The number registers **dn** and **dl** respectively contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in *nofill* mode regardless of the current *V*. Constant-spaced (**cs**) or emboldened (**bd**) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way to do this is to imbed in the diversion the appropriate **cs** or **bd** requests with the *transparent* mechanism described in §10.6.

Diversions may be nested and certain parameters and registers are associated with the current diversion level (the top non-diversion level may be thought of as the 0th diversion level). These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (see **mk** and **rt**), the current vertical place (**.d** register), the current high-water text base-line (**.h** register), and the current diversion name (**.z** register).

7.5. Traps. Three types of trap mechanisms are available—page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps may be planted using **wh** at any page position including the top. This trap position may be changed using **ch**. Trap positions at or below the bottom of the page have no effect unless or until moved to within the page or rendered effective by an increase in page length. Two traps may be planted at the *same* position only by first planting them at different positions and then moving one of the traps; the first planted trap will conceal the second unless and until the first one is moved (see Tutorial Examples §T5). If the first one is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line of text is output whose vertical size *reaches* or *sweeps past* the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the **.t** register; if there are no traps between the current position and the bottom of the page, the distance returned is the distance to the page bottom.

A macro-invocation trap effective in the current diversion may be planted using **dt**. The **.t** register works in a diversion; if there is no subsequent trap a *large* distance is returned. For a description of input-line-count traps, see **it** below.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.de *xx yy*	-	*.yy*=..	-	Define or redefine the macro *xx*. The contents of the macro begin on the next input line. Input lines are copied in *copy mode* until the definition is terminated by a line beginning with *.yy*, whereupon the macro *yy* is called. In the absence of *yy*, the definition is terminated by a line beginning with "..". A macro may contain **de** requests provided the terminating macros differ or the contained definition terminator is concealed. ".." can be concealed as \\.. which will copy as \.. and be reread as "..".
.am *xx yy*	-	*.yy*=..	-	Append to macro (append version of **de**).
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*. Any initial double-quote in *string* is stripped off to permit initial blanks.
.as *xx string*	-	ignored	-	Append *string* to string *xx* (append version of **ds**).
.rm *xx*	-	ignored	-	Remove request, macro, or string. The name *xx* is removed from the name list and any related storage space is freed. Subsequent references will have no effect.
.rn *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*. If *yy* exists, it is first removed.
.di *xx*	-	end	D	Divert output to macro *xx*. Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request **di** or **da** is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used.

.da *xx*	-	end	D	Divert, appending to *xx* (append version of **di**).
.wh *N xx*	-	-	v	Install a trap to invoke *xx* at page position *N;* a *negative N* will be interpreted with respect to the page *bottom.* Any macro previously planted at *N* is replaced by *xx.* A zero *N* refers to the *top* of a page. In the absence of *xx,* the first found trap at *N,* if any, is removed.
.ch *xx N*	-	-	v	Change the trap position for macro *xx* to be *N.* In the absence of *N,* the trap, if any, is removed.
.dt *N xx*	-	off	D,v	Install a diversion trap at position *N* in the *current* diversion to invoke macro *xx.* Another **dt** will redefine the diversion trap. If no arguments are given, the diversion trap is removed.
.it *N xx*	-	off	E	Set an input-line-count trap to invoke the macro *xx* after *N* lines of *text* input have been read (control or request lines don't count). The text may be in-line text or text interpolated by inline or trap-invoked macros.
.em *xx*	none	none	-	The macro *xx* will be invoked when all input has ended. The effect is the same as if the contents of *xx* had been at the end of the last file processed.

8. Number Registers

A variety of parameters are available to the user as predefined, named *number registers* (see Summary and Index, page 7). In addition, the user may define his own named registers. Register names are one or two characters long and *do not* conflict with request, macro, or string names. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired and may be used in numerical *expressions* (§1.4).

Number registers are created and modified using **nr**, which specifies the name, numerical value, and the auto-increment size. Registers are also modified, if accessed with an auto-incrementing sequence. If the registers *x* and *xx* both contain *N* and have the auto-increment size *M,* the following access sequences have the effect shown:

Sequence	Effect on Register	Value Interpolated
\n*x*	none	*N*
\n(*xx*	none	*N*
\n+*x*	*x* incremented by *M*	$N+M$
\n−*x*	*x* decremented by *M*	$N-M$
\n+(*xx*	*xx* incremented by *M*	$N+M$
\n−(*xx*	*xx* decremented by *M*	$N-M$

When interpolated, a number register is converted to decimal (default), decimal with leading zeros, lower-case Roman, upper-case Roman, lower-case sequential alphabetic, or upper-case sequential alphabetic according to the format specified by **af**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nr *R* ±*N M*		-	u	The number register *R* is assigned the value ±*N* with respect to the previous value, if any. The increment for auto-incrementing is set to *M.*

.af *R c*	arabic	-	-	Assign format *c* to register *R*. The available formats are:

Format	Numbering Sequence
1	0,1,2,3,4,5,...
001	000,001,002,003,004,005,...
i	0,i,ii,iii,iv,v,...
I	0,I,II,III,IV,V,...
a	0,a,b,c,...,z,aa,ab,...,zz,aaa,...
A	0,A,B,C,...,Z,AA,AB,...,ZZ,AAA,...

An arabic format having *N* digits specifies a field width of *N* digits (example 2 above). The read-only registers and the *width* function (§11.2) are always arabic.

.rr *R*	-	ignored	-	Remove register *R*. If many registers are being created dynamically, it may become necessary to remove no longer used registers to recapture internal storage space for newer registers.

9. Tabs, Leaders, and Fields

9.1. Tabs and leaders. The ASCII horizontal tab character and the ASCII SOH (hereafter known as the *leader* character) can both be used to generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal *tab stops* specifiable with **ta**. The default difference is that tabs generate motion and leaders generate a string of periods; **tc** and **lc** offer the choice of repeated character or motion. There are three types of internal tab stops—*left* adjusting, *right* adjusting, and *centering*. In the following table: *D* is the distance from the current position on the *input* line (where a tab or leader was found) to the next tab stop; *next-string* consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line; and *W* is the width of *next-string*.

Tab type	Length of motion or repeated characters	Location of *next-string*
Left	D	Following D
Right	$D-W$	Right adjusted within D
Centered	$D-W/2$	Centered on right end of D

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but may be used as *next-string* terminators.

Tabs and leaders are not interpreted in *copy mode*. \t and \a always generate a non-interpreted tab and leader respectively, and are equivalent to actual tabs and leaders in *copy mode*.

9.2. Fields. A *field* is contained between a *pair* of *field delimiter* characters, and consists of sub-strings separated by *padding* indicator characters. The field length is the distance on the *input* line from the position where the field begins to the next tab stop. The difference between the total length of all the sub-strings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is # and the padding indicator is ^, *#^xxx^right#* specifies a right-adjusted string with the string *xxx* centered in the remaining space.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ta *Nt* ...	0.8; 0.5in	none	E,m	Set tab stops and types. *t*=R, right adjusting; *t*=C, centering; *t* absent, left adjusting. TROFF tab stops are preset every 0.5in.; NROFF every 0.8in. The stop values are separated by spaces, and a value preceded by + is treated as an increment to the previous stop value.
.tc *c*	none	none	E	The tab repetition character becomes *c*, or is removed specifying motion.
.lc *c*	.	none	E	The leader repetition character becomes *c*, or is removed specifying motion.
.fc *a b*	off	off	-	The field delimiter is set to *a*; the padding indicator is set to the *space* character or to *b*, if given. In the absence of arguments the field mechanism is turned off.

10. Input and Output Conventions and Character Translations

10.1. Input character translations. Ways of inputting the graphic character set were discussed in §2.1. The ASCII control characters horizontal tab (§9.1), SOH (§9.1), and backspace (§10.3) are discussed elsewhere. The newline delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted, and may be used as delimiters or translated into a graphic with **tr** (§10.5). *All* others are ignored.

The *escape* character \ introduces *escape sequences*—causes the following character to mean another character, or to indicate some function. A complete list of such sequences is given in the Summary and Index on page 6. \ should not be confused with the ASCII control character ESC of the same name. The escape character \ can be input with the sequence \\. The escape character can be changed with **ec**, and all that has been said about the default \ becomes true for the new escape character. \e can be used to print whatever the current escape character is. If necessary or convenient, the escape mechanism may be turned off with **eo**, and restored with **ec**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ec *c*	\	\	-	Set escape character to \, or to *c*, if given.
.eo	on	-	-	Turn escape mechanism off.

10.2. Ligatures. Five ligatures are available in the current TROFF character set — **ﬁ**, **ﬂ**, **ﬀ**, **ﬃ**, and **ﬄ**. They may be input (even in NROFF) by \(fi, \(fl, \(ff, \(Fi, and \(Fl respectively. The ligature mode is normally on in TROFF, and *automatically* invokes ligatures during input.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.lg *N*	off; on	on	-	Ligature mode is turned on if *N* is absent or non-zero, and turned off if *N*=0. If *N*=2, only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request, macro, string, register, or file names, and in *copy mode*. No effect in NROFF.

10.3. Backspacing, underlining, overstriking, etc. Unless in *copy mode*, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character. Underlining as a form of line-drawing is discussed in §12.4. A generalized overstriking function is described in §12.1.

NROFF automatically underlines characters in the *underline* font, specifiable with **uf**, normally that on font position 2 (normally Times Italic, see §2.2). In addition to **ft** and \f*F*, the underline font may be selected by **ul** and **cu**. Underlining is restricted to an output-device-dependent subset of *reasonable* characters.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ul *N*	off	*N*=1	E	Underline in NROFF (italicize in TROFF) the next *N* input text lines. Actually, switch to *underline* font, saving the current font for later restoration; *other* font changes within the span of a **ul** will take effect, but the restoration will undo the last change. Output generated by **tl** (§14) *is* affected by the font change, but does *not* decrement *N*. If *N*>1, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching can prevent this.
.cu *N*	off	*N*=1	E	A variant of **ul** that causes *every* character to be underlined in NROFF. Identical to **ul** in TROFF.
.uf *F*	Italic	Italic	-	Underline font set to *F*. In NROFF, *F* may *not* be on position 1 (initially Times Roman).

10.4. Control characters. Both the control character . and the *no-break* control character ' may be changed, if desired. Such a change must be compatible with the design of any macros used in the span of the change, and particularly of any trap-invoked macros.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.cc *c*	.	.	E	The basic control character is set to *c*, or reset to ".".
.c2 *c*	'	'	E	The *nobreak* control character is set to *c*, or reset to "'".

10.5. Output translation. One character can be made a stand-in for another character using **tr**. All text processing (e. g. character comparisons) takes place with the input (stand-in) character which appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.tr *abcd....*	none	-	O	Translate *a* into *b*, *c* into *d*, etc. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from *input* to *output* time.

10.6. Transparent throughput. An input line beginning with a **\!** is read in *copy mode* and *transparently* output (without the initial **\!**); the text processor is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to imbed control lines in a macro created by a diversion.

10.7. Comments and concealed newlines. An uncomfortably long input line that must stay one line (e. g. a string definition, or nofilled text) can be split into many physical lines by ending all but the last one with the escape ****. The sequence ****(newline) is *always* ignored—except in a comment. Comments may be imbedded at the *end* of any line by prefacing them with **\"**. The newline at the end of a comment cannot be concealed. A line beginning with **\"** will appear as a blank line and behave like **.sp 1**; a comment can be on a line by itself by beginning the line with **.\"**.

11. Local Horizontal and Vertical Motions, and the Width Function

11.1. Local Motions. The functions **\v'***N***'** and **\h'***N***'** can be used for *local* vertical and horizontal motion respectively. The distance *N* may be negative; the *positive* directions are *rightward* and *downward*. A *local* motion is one contained *within* a line. To avoid unexpected vertical dislocations, it is necessary that the *net* vertical local motion within a word in filled text and otherwise within a line balance to zero. The above and certain other escape sequences providing local motion are summarized in the following table.

Vertical Local Motion	Effect in TROFF	NROFF
\v'*N*'	Move distance *N*	
\u \d \r	½ em up ½ em down 1 em up	½ line up ½ line down 1 line up

Horizontal Local Motion	Effect in TROFF	NROFF
\h'*N*' \(space) \0	Move distance *N* Unpaddable space-size space Digit-size space	
\\| \^	1/6 em space 1/12 em space	ignored ignored

As an example, E^2 could be generated by the sequence **E\s−2\v'−0.4m'2\v'0.4m'\s+2**; it should be noted in this example that the 0.4 em vertical motions are at the smaller size.

11.2. Width Function. The *width* function **\w'*string*'** generates the numerical width of *string* (in basic units). Size and font changes may be safely imbedded in *string*, and will not affect the current environment. For example, **.ti −\w'1. 'u** could be used to temporarily indent leftward a distance equal to the size of the string "**1. **".

The width function also sets three number registers. The registers **st** and **sb** are set respectively to the highest and lowest extent of *string* relative to the baseline; then, for example, the total *height* of the string is **\n(stu−\n(sbu**. In TROFF the number register **ct** is set to a value between 0 and 3: 0 means that all of the characters in *string* were short lower case characters without descenders (like e); 1 means that at least one character has a descender (like y); 2 means that at least one character is tall (like H); and 3 means that both tall characters and characters with descenders are present.

11.3. Mark horizontal place. The escape sequence **\k*x*** will cause the *current* horizontal position in the *input line* to be stored in register *x*. As an example, the construction **\kx*word*\h'|\nxu+2u'*word*** will embolden *word* by backing up to almost its beginning and overprinting it, resulting in ***word***.

12. Overstrike, Bracket, Line-drawing, and Zero-width Functions

12.1. Overstriking. Automatically centered overstriking of up to nine characters is provided by the *overstrike* function **\o'*string*'**. The characters in *string* overprinted with centers aligned; the total width is that of the widest character. *string* should *not* contain local vertical motion. As examples, **\o'e\''** produces é, and **\o'\(mo\(sl'** produces ∉.

12.2. Zero-width characters. The function **\z*c*** will output *c* without spacing over it, and can be used to produce left-aligned overstruck combinations. As examples, **\z\(ci\(pl** will produce ⊕, and **\(br\z\(rn\(ul\(br** will produce the smallest possible constructed box ▯.

12.3. Large Brackets. The Special Mathematical Font contains a number of bracket construction pieces (⎧ ⎪ ⎩ ⎫ ⎭ ⎨ ⎬ │ ⎣ ⎦ ⎡ ⎤) that can be combined into various bracket styles. The function **\b'*string*'** may be used to pile up vertically the characters in *string* (the first character on top and the last at the bottom); the characters are vertically separated by 1 em and the total pile is centered 1/2 em above the current baseline (½ line in NROFF). For example, **\b'\(lc\(lf'E\|\b'\(rc\(rf'\x'−0.5m'\x'0.5m'** produces $\left[\text{E}\right]$.

12.4. Line drawing. The function **\l'*Nc*'** will draw a string of repeated *c*'s towards the right for a distance *N*. (**\l** is \(lower case L). If *c* looks like a continuation of an expression for *N*, it may insulated from *N* with a **\&**. If *c* is not specified, the _ (baseline rule) is used (underline character in NROFF). If *N* is negative, a backward horizontal motion of size *N* is made *before* drawing the string. Any space resulting from *N*/(size of *c*) having a remainder is put at the beginning (left end) of the string. In the case of characters that are designed to be connected such as baseline-rule _, underrule _, and root-en ‾, the remainder space is covered by over-lapping. If *N* is *less* than the width of *c*, a single *c* is centered on a distance *N*. As an example, a macro to underscore a string can be written

```
.de us
\\$1\l'|0\(ul'
..
```

or one to draw a box around a string

```
        .de bx
        \(br\|\\$1\|\(br\l´|0\(rn´\l´|0\(ul´
        ..
```

such that

```
        .ul "underlined words"
```

and

```
        .bx "words in a box"
```

yield underlined words and words in a box.

The function \L´*Nc*´ will draw a vertical line consisting of the (optional) character *c* stacked vertically apart 1 em (1 line in NROFF), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the *box rule* | (\(br); the other suitable character is the *bold vertical* | (\(bv). The line is begun without any initial motion relative to the current base line. A positive *N* specifies a line drawn downward and a negative *N* specifies a line drawn upward. After the line is drawn *no* compensating motions are made; the instantaneous baseline is at the *end* of the line.

The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width *box-rule* and the ½-em wide *underrule* were *designed* to form corners when using 1-em vertical spacings. For example the macro

```
        .de eb
        .sp −1          \"compensate for next automatic base-line spacing
        .nf             \"avoid possibly overflowing word buffer
        \h´−.5n´\L´|\\nau−1´\l´\\n(.lu+1n\(ul´\L´−|\\nau+1´\l´|0u−.5n\(ul´    \"draw box
        .fi
        ..
```

will draw a box around some text whose beginning vertical place was saved in number register *a* (e. g. using .mk a) as done for this paragraph.

13. Hyphenation.

The automatic hyphenation may be switched off and on. When switched on with **hy**, several variants may be set. A *hyphenation indicator* character may be imbedded in a word to specify desired hyphenation points, or may be prepended to suppress hyphenation. In addition, the user may specify a small exception word list.

Only words that consist of a central alphabetic string surrounded by (usually null) non-alphabetic strings are considered candidates for automatic hyphenation. Words that were input containing hyphens (minus), em-dashes (\(em), or hyphenation indicator characters—such as mother-in-law—are *always* subject to splitting after those characters, whether or not automatic hyphenation is on or off.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nh	hyphenate	-	E	Automatic hyphenation is turned off.
.hy*N*	on,*N*=1	on,*N*=1	E	Automatic hyphenation is turned on for *N*≥1, or off for *N*=0. If *N*=2, *last* lines (ones that will cause a trap) are not hyphenated. For *N*=4 and 8, the last and first two characters respectively of a word are not split off. These values are additive; i. e. *N*=14 will invoke all three restrictions.
.hc *c*	\%	\%	E	Hyphenation indicator character is set to *c* or to the default \%. The indicator does not appear in the output.
.hw *word1* ...		ignored	-	Specify hyphenation points in words with imbedded minus signs. Versions of a word with terminal *s* are

implied; i. e. *dig−it* implies *dig−its*. This list is examined initially *and* after each suffix stripping. The space available is small—about 128 characters.

14. Three Part Titles.

The titling function **tl** provides for automatic placement of three fields at the left, center, and right of a line with a title-length specifiable with **lt**. **tl** may be used anywhere, and is independent of the normal text collecting process. A common use is in header and footer macros.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.tl *'left'center'right'*	-		-	The strings *left*, *center*, and *right* are respectively left-adjusted, centered, and right-adjusted in the current title-length. Any of the strings may be empty, and overlapping is permitted. If the page-number character (initially %) is found within any of the fields it is replaced by the current page number having the format assigned to register %. Any character may be used as the string delimiter.
.pc *c*	%	off	-	The page number character is set to *c*, or removed. The page-number register remains %.
.lt ±*N*	6.5 in	previous	E,m	Length of title set to ±*N*. The line-length and the title-length are *independent*. Indents do not apply to titles; page-offsets do.

15. Output Line Numbering.

Automatic sequence numbering of output lines may be requested with **nm**. When in effect, a
three-digit, arabic number plus a digit-space is prepended to output text lines. The text lines are
3 thus offset by four digit-spaces, and otherwise retain their line length; a reduction in line length
may be desired to keep the right margin aligned with an earlier margin. Blank lines, other vertical
spaces, and lines generated by **tl** are *not* numbered. Numbering can be temporarily suspended with
6 **nn**, or with an **.nm** followed by a later **.nm +0**. In addition, a line number indent *I*, and the
number-text separation *S* may be specified in digit-spaces. Further, it can be specified that only
those line numbers that are multiples of some number *M* are to be printed (the others will appear
9 as blank number fields).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nm ±*N M S I*		off	E	Line number mode. If ±*N* is given, line numbering is turned on, and the next output line numbered is numbered ±*N*. Default values are *M*=1, *S*=1, and *I*=0. Parameters corresponding to missing arguments are unaffected; a non-numeric argument is considered missing. In the absence of all arguments, numbering is turned off; the next line number is preserved for possible further use in number register **ln**.
.nn *N*	-	*N*=1	E	The next *N* text output lines are not numbered.

As an example, the paragraph portions of this section are numbered with *M*=3: **.nm 1 3** was
placed at the beginning; **.nm** was placed at the end of the first paragraph; and **.nm +0** was placed
12 in front of this paragraph; and **.nm** finally placed at the end. Line lengths were also changed (by
\w'0000'u) to keep the right side aligned. Another example is **.nm +5 5 x 3** which turns on
numbering with the line number of the next line to be 5 greater than the last numbered line, with
15 *M*=5, with spacing *S* untouched, and with the indent *I* set to 3.

16. Conditional Acceptance of Input

In the following, *c* is a one-character, built-in *condition* name, ! signifies *not*, *N* is a numerical expression, *string1* and *string2* are strings delimited by any non-blank, non-numeric character *not* in the strings, and *anything* represents what is conditionally accepted.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.if *c anything*	-	-		If condition *c* true, accept *anything* as input; in multi-line case use \\{*anything*\\}.
.if !*c anything*	-	-		If condition *c* false, accept *anything*.
.if *N anything*	-	u		If expression $N > 0$, accept *anything*.
.if !*N anything*	-	u		If expression $N \leqslant 0$, accept *anything*.
.if *'string1'string2' anything*		-		If *string1* identical to *string2*, accept *anything*.
.if !*'string1'string2' anything*		-		If *string1* not identical to *string2*, accept *anything*.
.ie *c anything*	-	u		If portion of if-else; all above forms (like **if**).
.el *anything*	-	-		Else portion of if-else.

The built-in condition names are:

Condition Name	True If
o	Current page number is odd
e	Current page number is even
t	Formatter is TROFF
n	Formatter is NROFF

If the condition *c* is *true*, or if the number *N* is greater than zero, or if the strings compare identically (including motions and character size and font), *anything* is accepted as input. If a ! precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multi-line case, the first line must begin with a left delimiter \\{ and the last line must end with a right delimiter \\}.

The request **ie** (if-else) is identical to **if** except that the acceptance state is remembered. A subsequent and matching **el** (else) request then uses the reverse sense of that state. **ie** - **el** pairs may be nested.

Some examples are:

```
        .if e .tl 'Even Page %'''
```

which outputs a title if the page number is even; and

```
        .ie \n%>1 \{\
        'sp 0.5i
        .tl 'Page %'''
        'sp |1.2i \}
        .el .sp |2.5i
```

which treats page 1 differently from other pages.

17. Environment Switching.

A number of the parameters that control the text processing are gathered together into an *environment*, which can be switched by the user. The environment parameters are those associated with requests noting E in their *Notes* column; in addition, partially collected lines and words are in the environment. Everything else is global; examples are page-oriented parameters, diversion-oriented parameters, number registers, and macro and string definitions. All environments are initialized with default parameter values.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ev *N*	*N*=0	previous	-	Environment switched to environment $0 \leqslant N \leqslant 2$. Switching is done in push-down fashion so that restoring a previous environment *must* be done with .ev rather than specific reference.

18. Insertions from the Standard Input

The input can be temporarily switched to the system *standard input* with **rd**, which will switch back when *two* newlines in a row are found (the *extra* blank line is not used). This mechanism is intended for insertions in form-letter-like documentation. On UNIX, the *standard input* can be the user's keyboard, a *pipe*, or a *file*.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.**rd** *prompt*	-	*prompt*=BEL	-	Read insertion from the standard input until two newlines in a row are found. If the standard input is the user's keyboard, *prompt* (or a BEL) is written onto the user's terminal. **rd** behaves like a macro, and arguments may be placed after *prompt*.
.**ex**	-	-	-	Exit from NROFF/TROFF. Text processing is terminated exactly as if all input had ended.

If insertions are to be taken from the terminal keyboard *while* output is being printed on the terminal, the command line option **−q** will turn off the echoing of keyboard input and prompt only with BEL. The regular input and insertion input *cannot* simultaneously come from the standard input.

As an example, multiple copies of a form letter may be prepared by entering the insertions for all the copies in one file to be used as the standard input, and causing the file containing the letter to reinvoke itself using **nx** (§19); the process would ultimately be ended by an **ex** in the insertion file.

19. Input/Output File Switching

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.**so** *filename*		-	-	Switch source file. The top input (file reading) level is switched to *filename*. The effect of an **so** encountered in a macro is not felt until the input level returns to the file level. When the new file ends, input is again taken from the original file. **so**'s may be nested.
.**nx** *filename*		end-of-file	-	Next file is *filename*. The current file is considered ended, and the input is immediately switched to *filename*.
.**pi** *program*		-	-	Pipe output to *program* (NROFF only). This request must occur *before* any printing occurs. No arguments are transmitted to *program*.

20. Miscellaneous

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.**mc** *c N*	-	off	E,m	Specifies that a *margin* character *c* appear a distance *N* to the right of the right margin after each non-empty text line (except those produced by **tl**). If the output line is too-long (as can happen in nofill mode) the character will

				be appended to the line. If *N* is not given, the previous *N* is used; the initial *N* is 0.2 inches in NROFF and 1 em in TROFF. The margin character used with this paragraph was a 12-point box-rule.
.tm *string*	-	newline	-	After skipping initial blanks, *string* (rest of the line) is read in *copy mode* and written on the user's terminal.
.ig *yy*	-	*.yy*=..	-	Ignore input lines. **ig** behaves exactly like **de** (§7) except that the input is discarded. The input is read in *copy mode*, and any auto-incremented registers will be affected.
.pm *t*	-	all	-	Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; if *t* is given, only the total of the sizes is printed. The sizes is given in *blocks* of 128 characters.
.fl	-	-	B	Flush output buffer. Used in interactive debugging to force output.

21. Output and Error Messages.

The output from **tm**, **pm**, and the prompt from **rd**, as well as various *error* messages are written onto UNIX's *standard message* output. The latter is different from the *standard output*, where NROFF formatted output goes. By default, both are written onto the user's terminal, but they can be independently redirected.

Various *error* conditions may occur during the operation of NROFF and TROFF. Certain less serious errors having only local impact do not cause processing to terminate. Two examples are *word overflow*, caused by a word that is too large to fit into the word buffer (in fill mode), and *line overflow*, caused by an output line that grew too large to fit in the line buffer; in both cases, a message is printed, the offending excess is discarded, and the affected word or line is marked at the point of truncation with a * in NROFF and a ☞ in TROFF. The philosophy is to continue processing, if possible, on the grounds that output useful for debugging may be produced. If a serious error occurs, processing terminates, and an appropriate message is printed. Examples are the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

TUTORIAL EXAMPLES

T1. Introduction

Although NROFF and TROFF have by design a syntax reminiscent of earlier text processors* with the intent of easing their use, it is almost always necessary to prepare at least a small set of macro definitions to describe most documents. Such common formatting needs as page margins and footnotes are deliberately not built into NROFF and TROFF. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations.

The examples to be discussed are intended to be useful and somewhat realistic, but won't necessarily cover all relevant contingencies. Explicit numerical parameters are used in the examples to make them easier to read and to illustrate typical values. In many cases, number registers would really be used to reduce the number of places where numerical information is kept, and to concentrate conditional parameter initialization like that which depends on whether TROFF or NROFF is being used.

T2. Page Margins

As discussed in §3, *header* and *footer* macros are usually defined to describe the top and bottom page margin areas respectively. A trap is planted at page position 0 for the header, and at $-N$ (N from the page bottom) for the footer. The simplest such definitions might be

```
.de hd          \"define header
'sp 1i
..              \"end definition
.de fo          \"define footer
'bp
..              \"end definition
.wh 0 hd
.wh −1i fo
```

which provide blank 1 inch top and bottom margins. The header will occur on the *first* page, only if the definition and trap exist prior to the initial pseudo-page transition (§3). In fill mode, the output line that springs the footer trap was typically forced out because some part or whole word didn't fit on it. If anything in the footer and header that follows causes a *break*, that word or part word will be forced out. In this and other examples, requests like **bp** and **sp** that normally cause breaks are invoked using the *no-break* control character ' to avoid this. When the header/footer design contains material requiring independent text processing, the environment may be switched, avoiding most interaction with the running text.

*For example: P. A. Crisman, Ed., *The Compatible Time-Sharing System*, MIT Press, 1965, Section AH9.01 (Description of RUNOFF program on MIT's CTSS system).

A more realistic example would be

```
.de hd              \"header
.if t .tl '\(rn''\(rn'  \"troff cut mark
.if \\n%>1 \{\
'sp |0.5i−1         \"tl base at 0.5i
.tl ''− % −''       \"centered page number
.ps                 \"restore size
.ft                 \"restore font
.vs  \}             \"restore vs
'sp |1.0i           \"space to 1.0i
.ns                 \"turn on no-space mode
..
.de fo              \"footer
.ps 10              \"set footer/header size
.ft R               \"set font
.vs 12p             \"set base-line spacing
.if \\n%=1 \{\
'sp |\\n(.pu−0.5i−1  \"tl base 0.5i up
.tl ''− % −'' \}  \"first page number
'bp
..
.wh 0 hd
.wh −1i fo
```

which sets the size, font, and base-line spacing for the header/footer material, and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. If TROFF is used, a *cut mark* is drawn in the form of *root-en*'s at each margin. The sp's refer to absolute positions to avoid dependence on the base-line spacing. Another reason for this in the footer is that the footer is invoked by printing a line whose vertical spacing swept past the trap position by possibly as much as the base-line spacing. The *no-space* mode is turned on at the end of **hd** to render ineffective accidental occurrences of **sp** at the top of the running text.

The above method of restoring size, font, etc. presupposes that such requests (that set *previous* value) are *not* used in the running text. A better scheme is save and restore both the current *and* previous values as shown for size in the following:

```
.de fo
.nr s1 \\n(.s    \"current size
.ps
.nr s2 \\n(.s    \"previous size
. ---            \"rest of footer
..
.de hd
. ---            \"header stuff
.ps \\n(s2       \"restore previous size
.ps \\n(s1       \"restore current size
..
```

Page numbers may be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn           \"bottom number
.tl ''- % -''    \"centered page number
..
.wh −0.5i−1v bn \"tl base 0.5i up
```

T3. Paragraphs and Headings

The housekeeping associated with starting a new paragraph should be collected in a paragraph macro that, for example, does the desired preparagraph spacing, forces the correct font, size, base-line spacing, and indent, checks that enough space remains for *more than one* line, and requests a temporary indent.

```
.de pg           \"paragraph
.br              \"break
.ft R            \"force font,
.ps 10           \"size,
.vs 12p          \"spacing,
.in 0            \"and indent
.sp 0.4          \"prespace
.ne 1+\\n(.Vu \"want more than 1 line
.ti 0.2i         \"temp indent
..
```

The first break in **pg** will force out any previous partial lines, and must occur before the **vs**. The forcing of font, etc. is partly a defense against prior error and partly to permit things like section heading macros to set parameters only once. The prespacing parameter is suitable for TROFF; a larger space, at least as big as the output device vertical resolution, would be more suitable in NROFF. The choice of remaining space to test for in the **ne** is the smallest amount greater than one line (the .V is the available vertical resolution).

A macro to automatically number section headings might look like:

```
.de sc           \"section
. ---            \"force font, etc.
.sp 0.4          \"prespace
.ne 2.4+\\n(.Vu \"want 2.4+ lines
.fi
\\n+S.
..
.nr S 0 1        \"init S
```

The usage is .sc, followed by the section heading text, followed by .pg. The **ne** test value includes one line of heading, 0.4 line in the following **pg**, and one line of the paragraph text. A word consisting of the next section number and a period is produced to begin the heading line. The format of the number may be set by **af** (§8).

Another common form is the labeled, indented paragraph, where the label protrudes left into the indent space.

```
.de lp           \"labeled paragraph
.pg
.in 0.5i         \"paragraph indent
.ta 0.2i 0.5i    \"label, paragraph
.ti 0
\t\\$1\t\c       \"flow into paragraph
..
```

The intended usage is ".lp *label*"; *label* will begin at 0.2 inch, and cannot exceed a length of 0.3 inch without intruding into the paragraph. The label could be right adjusted against 0.4 inch by setting the tabs instead with **.ta 0.4iR 0.5i**. The last line of **lp** ends with \c so that it will become a part of the first line of the text that follows.

T4. Multiple Column Output

The production of multiple column pages requires the footer macro to decide whether it was invoked by other than the last column, so that it will begin a new column rather than produce the bottom margin. The header can initialize a column register that the footer will increment and test. The following is arranged for two columns, but is easily modified for more.

```
.de hd          \"header
. ---
.nr cl 0 1      \"init column count
.mk             \"mark top of text
..
.de fo          \"footer
.ie \\n+(cl<2 \{\
.po +3.4i       \"next column; 3.1+0.3
.rt             \"back to mark
.ns \}          \"no-space mode
.el \{\
.po \\nMu       \"restore left margin
. ---
'bp \}
..
.ll 3.1i        \"column width
.nr M \\n(.o    \"save left margin
```

Typically a portion of the top of the first page contains full width text; the request for the narrower line length, as well as another .mk would be made where the two column output was to begin.

T5. Footnote Processing

The footnote mechanism to be described is used by imbedding the footnotes in the input text at the point of reference, demarcated by an initial .fn and a terminal .ef:

```
.fn
Footnote text and control lines...
.ef
```

In the following, footnotes are processed in a separate environment and diverted for later printing in the space immediately prior to the bottom margin. There is provision for the case where the last collected footnote doesn't completely fit in the available space.

```
.de hd          \"header
. ---
.nr x 0 1       \"init footnote count
.nr y 0-\\nb    \"current footer place
.ch fo -\\nbu   \"reset footer trap
.if \\n(dn .fz  \"leftover footnote
..
.de fo          \"footer
.nr dn 0        \"zero last diversion size
.if \\nx \{\
.ev 1           \"expand footnotes in ev1
.nf             \"retain vertical size
.FN             \"footnotes
.rm FN          \"delete it
.if "\\n(.z"fy" .di \"end overflow diversion
.nr x 0         \"disable fx
.ev \}          \"pop environment
. ---
'bp
..
.de fx          \"process footnote overflow
.if \\nx .di fy \"divert overflow
..
.de fn          \"start footnote
.da FN          \"divert (append) footnote
.ev 1           \"in environment 1
.if \\n+x=1 .fs \"if first, include separator
.fi             \"fill mode
..
.de ef          \"end footnote
.br             \"finish output
.nr z \\n(.v    \"save spacing
.ev             \"pop ev
.di             \"end diversion
.nr y -\\n(dn \"new footer position,
.if \\nx=1 .nr y -(\\n(.v-\\nz) \
                \"uncertainty correction
.ch fo \\nyu    \"y is negative
.if (\\n(nl+1v)>(\\n(.p+\\ny) \
.ch fo \\n(nlu+1v \"it didn't fit
..
.de fs          \"separator
\l'1i'          \"1 inch rule
.br
..
.de fz          \"get leftover footnote
.fn
.nf             \"retain vertical size
.fy             \"where fx put it
.ef
..
.nr b 1.0i      \"bottom margin size
.wh 0 hd        \"header trap
.wh 12i fo      \"footer trap, temp position
.wh -\\nbu fx \"fx at footer position
.ch fo -\\nbu   \"conceal fx with fo
```

The header hd initializes a footnote count register x, and sets both the current footer trap position register y and the footer trap itself to a nominal position specified in register b. In addition, if the register dn indicates a leftover footnote, fz is invoked to reprocess it. The footnote start macro fn begins a diversion (append) in environment 1, and increments the count x; if the count 's one, the footnote separator fs is interpolated. The separator is kept in a separate macro to permit user redefinition. The footnote end macro ef restores the previous environment and ends the diversion after saving the spacing size in register z. y is then decremented by the size of the footnote, available in **dn**; then on the first footnote, y is further decremented by the difference in vertical base-line spacings of the two environments, to prevent the late triggering the footer trap from causing the last line of the combined footnotes to overflow. The footer trap is then set to the lower (on the page) of y or the current page position (**nl**) plus one line, to allow for printing the reference line. If indicated by **x**, the footer **fo** rereads the footnotes from **FN** in nofill mode in environment 1, and deletes **FN**. If the footnotes were too large to fit, the macro **fx** will be trap-invoked to redivert the overflow into **fy**, and the register **dn** will later indicate to the header whether **fy** is empty. Both **fo** and **fx** are planted in the nominal footer trap position in an order that causes **fx** to be concealed unless the **fo** trap is moved. The footer then terminates the overflow diversion, if necessary, and zeros **x** to disable **fx**, because the uncertainty correction together with a not-too-late triggering of the footer can result in the footnote rereading finishing before reaching the **fx** trap.

A good exercise for the student is to combine the multiple-column and footnote mechanisms.

T6. The Last Page

After the last input file has ended, NROFF and TROFF invoke the *end macro* (§7), if any, and when it finishes, eject the remainder of the page. During the eject, any traps encountered are processed normally. At the *end* of this last page, processing terminates *unless* a partial line, word, or partial word remains. If it is desired that another page be started, the end-macro

```
.de en              \"end-macro
\c
'bp
..
.em en
```

will deposit a null partial word, and effect another last page.

Table I

Font Style Examples

The following fonts are printed in 12-point, with a vertical spacing of 14-point, and with non-alphanumeric characters separated by ¼ em space. The Special Mathematical Font was specially prepared for Bell Laboratories by Graphic Systems, Inc. of Hudson, New Hampshire. The Times Roman, Italic, and Bold are among the many standard fonts available from that company.

Times Roman

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
● □ — - _ ¼ ½ ¾ ﬁ ﬂ ﬀ ﬃ ﬄ ° † ′ ¢ ® ©

Times Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
*! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |*
● □ — - _ ¼ ½ ¾ ﬁ ﬂ ﬀ ﬃ ﬄ ° † ′ ¢ ® ©

Times Bold

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
● □ — - _ ¼ ½ ¾ ﬁ ﬂ ﬀ ﬃ ﬄ ° † ′ ¢ ® ©

Special Mathematical Font

" ´ \ ˆ _ ` ‾ / < > { } # @ + − = ∗
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ ς τ υ φ χ ψ ω
Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω
√ ‾ ≥ ≤ ≡ ∼ ≃ ≠ → ← ↑ ↓ × ÷ ± ∪ ∩ ⊂ ⊃ ⊆ ⊇ ∞ ∂
§ ∇ ¬ ∫ ∝ ∅ ∈ ‡ ☞ ☜ 🔔 | ○ ⎛ ⎝ ⎠ ⎞ ⎩ ⎧ ⎫ ⎭ ⎪ ⎣ ⎦ ⎡ ⎤ │

Table II

Input Naming Conventions for ′, `,and − and for Non-ASCII Special Characters

Non-ASCII characters and *minus* on the standard fonts.

Char	*Input Name*	*Character Name*	*Char*	*Input Name*	*Character Name*
’	′	close quote	fi	\(fi	fi
‘	`	open quote	fl	\(fl	fl
—	\(em	3/4 Em dash	ff	\(ff	ff
-	-	hyphen or	ffi	\(Fi	ffi
-	\(hy	hyphen	ffl	\(Fl	ffl
−	\-	current font minus	°	\(de	degree
●	\(bu	bullet	†	\(dg	dagger
□	\(sq	square	′	\(fm	foot mark
_	\(ru	rule	¢	\(ct	cent sign
¼	\(14	1/4	®	\(rg	registered
½	\(12	1/2	©	\(co	copyright
¾	\(34	3/4			

Non-ASCII characters and ′, `, _, +, −, =, and ∗ on the special font.

The ASCII characters @, #, ", ′, `, <, >, \, {, }, ~, ^, and _ exist *only* on the special font and are printed as a 1-em space if that font is not mounted. The following characters exist only on the special font except for the upper case Greek letter names followed by † which are mapped into upper case English letters in whatever font is mounted on font position one (default Times Roman). The special math plus, minus, and equals are provided to insulate the appearance of equations from the choice of standard fonts.

Char	*Input Name*	*Character Name*	*Char*	*Input Name*	*Character Name*
+	\(pl	math plus	κ	\(*k	kappa
−	\(mi	math minus	λ	\(*l	lambda
=	\(eq	math equals	μ	\(*m	mu
∗	\(**	math star	ν	\(*n	nu
§	\(sc	section	ξ	\(*c	xi
´	\(aa	acute accent	ο	\(*o	omicron
`	\(ga	grave accent	π	\(*p	pi
_	\(ul	underrule	ρ	\(*r	rho
/	\(sl	slash (matching backslash)	σ	\(*s	sigma
α	\(*a	alpha	ς	\(ts	terminal sigma
β	\(*b	beta	τ	\(*t	tau
γ	\(*g	gamma	υ	\(*u	upsilon
δ	\(*d	delta	φ	\(*f	phi
ε	\(*e	epsilon	χ	\(*x	chi
ζ	\(*z	zeta	ψ	\(*q	psi
η	\(*y	eta	ω	\(*w	omega
θ	\(*h	theta	Α	\(*A	Alpha†
ι	\(*i	iota	Β	\(*B	Beta†

Char	Input Name	Character Name
Γ	\(*G	Gamma
Δ	\(*D	Delta
Ε	\(*E	Epsilon†
Ζ	\(*Z	Zeta†
Η	\(*Y	Eta†
Θ	\(*H	Theta
Ι	\(*I	Iota†
Κ	\(*K	Kappa†
Λ	\(*L	Lambda
Μ	\(*M	Mu†
Ν	\(*N	Nu†
Ξ	\(*C	Xi
Ο	\(*O	Omicron†
Π	\(*P	Pi
Ρ	\(*R	Rho†
Σ	\(*S	Sigma
Τ	\(*T	Tau†
Υ	\(*U	Upsilon
Φ	\(*F	Phi
Χ	\(*X	Chi†
Ψ	\(*Q	Psi
Ω	\(*W	Omega
√	\(sr	square root
‾	\(rn	root en extender
≥	\(>=	>=
≤	\(<=	<=
≡	\(==	identically equal
≃	\(~=	approx =
~	\(ap	approximates
≠	\(!=	not equal
→	\(->	right arrow
←	\(<-	left arrow
↑	\(ua	up arrow
↓	\(da	down arrow
×	\(mu	multiply
÷	\(di	divide
±	\(+-	plus-minus
∪	\(cu	cup (union)
∩	\(ca	cap (intersection)
⊂	\(sb	subset of
⊃	\(sp	superset of
⊆	\(ib	improper subset
⊇	\(ip	improper superset
∞	\(if	infinity
∂	\(pd	partial derivative
∇	\(gr	gradient
¬	\(no	not
∫	\(is	integral sign
∝	\(pt	proportional to
∅	\(es	empty set
∈	\(mo	member of

Char	Input Name	Character Name
│	\(br	box vertical rule
‡	\(dd	double dagger
☞	\(rh	right hand
☜	\(lh	left hand
🔔	\(bs	Bell System logo
∣	\(or	or
○	\(ci	circle
⎧	\(lt	left top of big curly bracket
⎩	\(lb	left bottom
⎫	\(rt	right top
⎭	\(rb	right bot
⎨	\(lk	left center of big curly bracket
⎬	\(rk	right center of big curly bracket
⎪	\(bv	bold vertical
⎣	\(lf	left floor (left bottom of big square bracket)
⎦	\(rf	right floor (right bottom)
⎡	\(lc	left ceiling (left top)
⎤	\(rc	right ceiling (right top)

May 15, 1977

Summary of Changes to N/TROFF Since October 1976 Manual

Options

-h	(Nroff only) Output tabs used during horizontal spacing to speed output as well as reduce output byte count. Device tab settings assumed to be every 8 nominal character widths. The default settings of input (logical) tabs is also initialized to every 8 nominal character widths.
-z	Efficiently suppresses formatted output. Only message output will occur (from "tm"s and diagnostics).

Old Requests

.ad c	The adjustment type indicator "c" may now also be a number previously obtained from the ".j" register (see below).
.so name	The contents of file "name" will be interpolated at the point the "so" is encountered. Previously, the interpolation was done upon return to the file-reading input level.

New Request

.ab text — Prints "text" on the message output and terminates without further processing. If "text" is missing, "User Abort." is printed. Does not cause a break. The output buffer is flushed.

.fz F N — forces font "F" to be in size N. N may have the form N, +N, or -N. For example,

```
    .fz 3 -2
```

will cause an implicit \s-2 every time font 3 is entered, and a corresponding \s+2 when it is left. Special font characters occurring during the reign of font F will have the same size modification. If special characters are to be treated differently,

```
    .fz S F N
```

may be used to specify the size treatment of special characters during font F. For example,

```
    .fz 3 -3
    .fz S 3 -0
```

will cause automatic reduction of font 3 by 3 points while the special characters would not be affected. Any ".fp" request specifying a font on some position must precede ".fz" requests relating to that position.

New Predefined Number Registers.

.k	Read-only. Contains the horizontal size of the text portion (without indent) of the current partially collected output line, if any, in the current environment.
.j	Read-only. A number representing the current adjustment mode and type. Can be saved and later given to the "ad" request to restore a previous mode.
.P	Read-only. 1 if the current page is being printed, and zero otherwise.
.L	Read-only. Contains the current line-spacing parameter ("ls").
c.	General register access to the input line-number in the current input file. Contains the same value as the read-only ".c" register.

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

troff is a text-formatting program for driving the Graphic Systems phototypesetter on the UNIX† and GCOS operating systems. This device is capable of producing high quality text; this paper is an example of **troff** output.

The phototypesetter itself normally runs with four fonts, containing roman, italic and bold letters (as on this page), a full greek alphabet, and a substantial number of special characters and mathematical symbols. Characters can be printed in a range of sizes, and placed anywhere on the page.

troff allows the user full control over fonts, sizes, and character positions, as well as the usual features of a formatter — right-margin justification, automatic hyphenation, page titling and numbering, and so on. It also provides macros, arithmetic variables and operations, and conditional testing, for complicated formatting tasks.

This document is an introduction to the most basic use of **troff**. It presents just enough information to enable the user to do simple formatting tasks like making viewgraphs, and to make incremental changes to existing packages of **troff** commands. In most respects, the UNIX formatter **nroff** is identical to **troff**, so this document also serves as a tutorial on **nroff**.

August 4, 1978

†UNIX is a Trademark of Bell Laboratories.

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

troff [1] is a text-formatting program, written by J. F. Ossanna, for producing high-quality printed output from the phototypesetter on the UNIX and GCOS operating systems. This document is an example of **troff** output.

The single most important rule of using **troff** is not to use it directly, but through some intermediary. In many ways, **troff** resembles an assembly language — a remarkably powerful and flexible one — but nonetheless such that many operations must be specified at a level of detail and in a form that is too hard for most people to use effectively.

For two special applications, there are programs that provide an interface to **troff** for the majority of users. **eqn** [2] provides an easy to learn language for typesetting mathematics; the **eqn** user need know no **troff** whatsoever to typeset mathematics. **tbl** [3] provides the same convenience for producing tables of arbitrary complexity.

For producing straight text (which may well contain mathematics or tables), there are a number of 'macro packages' that define formatting rules and operations for specific styles of documents, and reduce the amount of direct contact with **troff**. In particular, the '−ms' [4] and PWB/MM [5] packages for Bell Labs internal memoranda and external papers provide most of the facilities needed for a wide range of document preparation. (This memo was prepared with '−ms'.) There are also packages for viewgraphs, for simulating the older **roff** formatters on UNIX and GCOS, and for other special applications. Typically you will find these packages easier to use than **troff** once you get beyond the most trivial operations; you should always consider them first.

In the few cases where existing packages don't do the whole job, the solution is *not* to write an entirely new set of **troff** instructions from scratch, but to make small changes to adapt packages that already exist.

In accordance with this philosophy of letting someone else do the work, the part of **troff** described here is only a small part of the whole, although it tries to concentrate on the more useful parts. In any case, there is no attempt to be complete. Rather, the emphasis is on showing how to do simple things, and how to make incremental changes to what already exists. The contents of the remaining sections are:

The **troff** described here is the C-language version running on UNIX at Murray Hill, as documented in [1].

To use **troff** you have to prepare not only the actual text you want printed, but some information that tells *how* you want it printed. (Readers who use **roff** will find the approach familiar.) For **troff** the text and the formatting information are often intertwined quite intimately. Most commands to **troff** are placed on a line separate from the text itself, beginning with a period (one command per line). For example,

```
Some text.
.ps 14
Some more text.
```

will change the 'point size', that is, the size of the letters being printed, to '14 point' (one point is 1/72 inch) like this:

Some text. Some more text.

Occasionally, though, something special occurs in the middle of a line — to produce

Area = πr^2

you have to type

```
Area = \(*p\fIr\fR\|\s8\u2\d\s0
```

(which we will explain shortly). The backslash character \ is used to introduce **troff** commands and special characters within a line of text.

2. Point Sizes; Line Spacing

As mentioned above, the command .ps sets the point size. One point is 1/72 inch, so 6-point characters are at most 1/12 inch high, and 36-point characters are ½ inch. There are 15 point sizes, listed below.

6 point: Pack my box with five dozen liquor jugs.
7 point: Pack my box with five dozen liquor jugs.
8 point: Pack my box with five dozen liquor jugs.
9 point: Pack my box with five dozen liquor jugs.
10 point: Pack my box with five dozen liquor
11 point: Pack my box with five dozen
12 point: Pack my box with five dozen
14 point: Pack my box with five
16 point 18 point 20 point
22 24 28 36

If the number after .ps is not one of these legal sizes, it is rounded up to the next valid value, with a maximum of 36. If no number follows .ps, **troff** reverts to the previous size, whatever it was. **troff** begins with point size 10, which is usually fine. This document is in 9 point.

The point size can also be changed in the middle of a line or even a word with the in-line command \s. To produce

UNIX runs on a PDP-11/45

type

```
\s8UNIX\s10 runs on a \s8PDP-\s1011/45
```

As above, \s should be followed by a legal point size, except that \s0 causes the size to revert to its previous value. Notice that \s1011 can be understood correctly as 'size 10, followed by an 11', if the size is legal, but not otherwise. Be cautious with similar constructions.

Relative size changes are also legal and useful:

```
\s−2UNIX\s+2
```

temporarily decreases the size, whatever it is, by two points, then restores it. Relative size changes have the advantage that the size difference is independent of the starting size of the document. The amount of the relative change is restricted to a single digit.

The other parameter that determines what the type looks like is the spacing between lines, which is set independently of the point size. Vertical spacing is measured from the bottom of one line to the bottom of the next. The command to control vertical spacing is .vs. For running text, it is usually best to set the vertical spacing about 20% bigger than the character size. For example, so far in this document, we have used "9 on 11", that is,

```
.ps 9
.vs 11p
```

If we changed to

```
.ps 9
.vs 9p
```

the running text would look like this. After a few lines, you will agree it looks a little cramped. The right vertical spacing is partly a matter of taste, depending on how much text you want to squeeze into a given space, and partly a matter of traditional printing style. By default, **troff** uses 10 on 12.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. This is 12 on 14.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. For example, 10 on 12 uses about twice as much space as 7 on 8. This is 6 on 7, which is even smaller. It packs a lot more words per line, but you can go blind trying to read it.

When used without arguments, .ps and .vs revert to the previous size and vertical spacing respectively.

The command .sp is used to get extra vertical space. Unadorned, it gives you one extra blank line (one .vs, whatever that has been set to). Typically, that's more or less than you want, so .sp can be followed by information about how much space you want —

```
.sp 2i
```

means 'two inches of vertical space'.

```
.sp 2p
```

means 'two points of vertical space'; and

```
.sp 2
```

means 'two vertical spaces' — two of whatever

.vs is set to (this can also be made explicit with .sp 2v); **troff** also understands decimal fractions in most places, so

```
.sp 1.5i
```

is a space of 1.5 inches. These same scale factors can be used after .vs to define line spacing, and in fact after most commands that deal with physical dimensions.

It should be noted that all size numbers are converted internally to 'machine units', which are 1/432 inch (1/6 point). For most purposes, this is enough resolution that you don't have to worry about the accuracy of the representation. The situation is not quite so good vertically, where resolution is 1/144 inch (1/2 point).

3. Fonts and Special Characters

troff and the typesetter allow four different fonts at any one time. Normally three fonts (Times roman, italic and bold) and one collection of special characters are permanently mounted.

abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

The greek, mathematical symbols and miscellany of the special font are listed in Appendix A.

troff prints in roman unless told otherwise. To switch into bold, use the .ft command

```
.ft B
```

and for italics,

```
.ft I
```

To return to roman, use .ft R; to return to the previous font, whatever it was, use either .ft P or just .ft. The 'underline' command

```
.ul
```

causes the next input line to print in italics. .ul can be followed by a count to indicate that more than one line is to be italicized.

Fonts can also be changed within a line or word with the in-line command \f:

bold*face* text

is produced by

```
\fBbold\fIface\fR text
```

If you want to do this so the previous font, whatever it was, is left undisturbed, insert extra \fP commands, like this:

```
\fBbold\fP\fIface\fP\fR text\fP
```

Because only the immediately previous font is remembered, you have to restore the previous font after each change or you can lose it. The same is true of .ps and .vs when used without an argument.

There are other fonts available besides the standard set, although you can still use only four at any given time. The command .fp tells **troff** what fonts are physically mounted on the typesetter:

```
.fp 3 H
```

says that the Helvetica font is mounted on position 3. (For a complete list of fonts and what they look like, see the **troff** manual.) Appropriate .fp commands should appear at the beginning of your document if you do not use the standard fonts.

It is possible to make a document relatively independent of the actual fonts used to print it by using font numbers instead of names; for example, \f3 and .ft~3 mean 'whatever font is mounted at position 3', and thus work for any setting. Normal settings are roman font on 1, italic on 2, bold on 3, and special on 4.

There is also a way to get 'synthetic' bold fonts by overstriking letters with a slight offset. Look at the .bd command in [1].

Special characters have four-character names beginning with \(, and they may be inserted anywhere. For example,

$$\frac{1}{4} + \frac{1}{2} = \frac{3}{4}$$

is produced by

```
\(14 + \(12 = \(34
```

In particular, greek letters are all of the form \(*-, where - is an upper or lower case roman letter reminiscent of the greek. Thus to get

$$\Sigma(\alpha\times\beta) \rightarrow \infty$$

in bare **troff** we have to type

```
\(*S(\(*a\(mu\(*b) \(-> \(if
```

That line is unscrambled as follows:

\(*S	Σ
(	(
\(*a	α
\(mu	$\times$
\(*b	β
)	)
\(->	$\rightarrow$
\(if	∞

A complete list of these special names occurs in Appendix A.

In **eqn** [2] the same effect can be achieved with the input

```
SIGMA ( alpha times beta ) -> inf
```

which is less concise, but clearer to the uninitiated.

Notice that each four-character name is a single character as far as **troff** is concerned — the 'translate' command

```
.tr \(mi\(em
```

is perfectly clear, meaning

```
.tr -—
```

that is, to translate − into —.

Some characters are automatically translated into others: grave ` and acute ´ accents (apostrophes) become open and close single quotes '‾'; the combination of "..." is generally preferable to the double quotes "...". Similarly a typed minus sign becomes a hyphen -. To print an explicit − sign, use \-. To get a backslash printed, use \e.

4. Indents and Line Lengths

troff starts with a line length of 6.5 inches, too wide for 8½×11 paper. To reset the line length, use the .ll command, as in

```
.ll 6i
```

As with .sp, the actual length can be specified in several ways; inches are probably the most intuitive.

The maximum line length provided by the typesetter is 7.5 inches, by the way. To use the full width, you will have to reset the default physical left margin ("page offset"), which is normally slightly less than one inch from the left edge of the paper. This is done by the .po command.

```
.po 0
```

sets the offset as far to the left as it will go.

The indent command .in causes the left margin to be indented by some specified amount from the page offset. If we use .in to move the left margin in, and .ll to move the right margin to the left, we can make offset blocks of text:

```
.in 0.3i
.ll −0.3i
text to be set into a block
.ll +0.3i
.in −0.3i
```

will create a block that looks like this:

> Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Notice the use of '+' and '−' to specify the amount of change. These change the previous setting by the specified amount, rather than just overriding it. The distinction is quite important: .ll +1i makes lines one inch longer; .ll 1i makes them one inch *long*.

With .in, .ll and .po, the previous value is used if no argument is specified.

To indent a single line, use the 'temporary indent' command .ti. For example, all paragraphs in this memo effectively begin with the command

```
.ti 3
```

Three of what? The default unit for .ti, as for most horizontally oriented commands (.ll, .in, .po), is ems; an em is roughly the width of the letter 'm' in the current point size. (Precisely, a em in size *p* is *p* points.) Although inches are usually clearer than ems to people who don't set type for a living, ems have a place: they are a measure of size that is proportional to the current point size. If you want to make text that keeps its proportions regardless of point size, you should use ems for all dimensions. Ems can be specified as scale factors directly, as in .ti 2.5m.

Lines can also be indented negatively if the indent is already positive:

```
.ti −0.3i
```

causes the next line to be moved back three tenths of an inch. Thus to make a decorative initial capital, we indent the whole paragraph, then move the letter 'P' back with a .ti command:

> Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Of course, there is also some trickery to make the 'P' bigger (just a '\s36P\s0'), and to move it down from its normal position (see the section on local motions).

5. Tabs

Tabs (the ASCII 'horizontal tab' character) can be used to produce output in columns, or to set the horizontal position of output. Typically tabs are used only in unfilled text. Tab stops are set by default every half inch from the current indent, but can be changed by the .ta command. To set stops every inch, for example,

```
.ta 1i 2i 3i 4i 5i 6i
```

Unfortunately the stops are left-justified only (as on a typewriter), so lining up columns of right-justified numbers can be painful. If you have many numbers, or if you need more complicated table layout, *don't* use **troff** directly; use the **tbl** program described in [3].

For a handful of numeric columns, you can do it this way: Precede every number by enough blanks to make it line up when typed.

```
.nf
.ta 1i 2i 3i
  1 tab   2 tab   3
 40 tab  50 tab  60
700 tab 800 tab 900
.fi
```

Then change each leading blank into the string \0. This is a character that does not print, but that has the same width as a digit. When printed, this will produce

```
  1        2        3
 40       50       60
700      800      900
```

It is also possible to fill up tabbed-over space with some character other than blanks by setting the 'tab replacement character' with the .tc command:

```
.ta 1.5i 2.5i
.tc \(ru       (\(ru is "_")
Name tab Age tab
```

produces

```
Name ________________ Age __________
```

To reset the tab replacement character to a blank, use .tc with no argument. (Lines can also be drawn with the \l command, described in Section 6.)

troff also provides a very general mechanism called 'fields' for setting up complicated columns. (This is used by **tbl**). We will not go into it in this paper.

6. Local Motions: Drawing lines and characters

Remember 'Area = πr^2' and the big 'P' in the Paternoster. How are they done? **troff** provides a host of commands for placing characters of any size at any place. You can use them to draw special characters or to tune your output for a particular appearance. Most of these commands are straightforward, but messy to read and tough to type correctly.

If you won't use **eqn**, subscripts and superscripts are most easily done with the half-line local motions \u and \d. To go back up the page half a point-size, insert a \u at the desired place; to go down, insert a \d. (\u and \d should always be used in pairs, as explained below.) Thus

```
Area = \(*pr\u2\d
```

produces

Area = πr^2

To make the '2' smaller, bracket it with \s−2...\s0. Since \u and \d refer to the current point size, be sure to put them either both inside or both outside the size changes, or you will get an unbalanced vertical motion.

Sometimes the space given by \u and \d isn't the right amount. The \v command can be used to request an arbitrary amount of vertical motion. The in-line command

```
\v'(amount)'
```

causes motion up or down the page by the amount specified in '(amount)'. For example, to move the 'P' down, we used

```
.in +0.6i       (move paragraph in)
.ll −0.3i       (shorten lines)
.ti −0.3i       (move P back)
\v'2'\s36P\s0\v'−2'ater noster qui est
in caelis ...
```

A minus sign causes upward motion, while no sign or a plus sign means down the page. Thus \v'−2' causes an upward vertical motion of two line spaces.

There are many other ways to specify the amount of motion —

```
\v'0.1i'
\v'3p'
\v'−0.5m'
```

and so on are all legal. Notice that the scale specifier i or p or m goes inside the quotes. Any character can be used in place of the quotes; this is also true of all other **troff** commands described in this section.

Since **troff** does not take within-the-line vertical motions into account when figuring out where it is on the page, output lines can have unexpected positions if the left and right ends aren't at the same vertical position. Thus \v, like \u and \d, should always balance upward vertical motion in a line with the same amount in the downward direction.

Arbitrary horizontal motions are also available — \h is quite analogous to \v, except that the default scale factor is ems instead of line spaces. As an example,

```
\h'−0.1i'
```

causes a backwards motion of a tenth of an inch. As a practical matter, consider printing the mathematical symbol '>>'. The default spacing is too wide, so eqn replaces this by

```
>\h'−0.3m'>
```

to produce >>.

Frequently \h is used with the 'width function' \w to generate motions equal to the width of some character string. The construction

```
\w'thing'
```

is a number equal to the width of 'thing' in machine units (1/432 inch). All **troff** computations are ultimately done in these units. To move horizontally the width of an 'x', we can say

```
\h'\w'x'u'
```

As we mentioned above, the default scale factor for all horizontal dimensions is m, ems, so here we must have the u for machine units, or the motion produced will be far too large. **troff** is quite happy with the nested quotes, by the way, so long as you don't leave any out.

As a live example of this kind of construction, all of the command names in the text, like **.sp**, were done by overstriking with a slight offset. The commands for **.sp** are

```
.sp\h'−\w'.sp'u'\h'1u'.sp
```

That is, put out '.sp', move left by the width of '.sp', move right 1 unit, and print '.sp' again. (Of course there is a way to avoid typing that much input for each command name, which we will discuss in Section 11.)

There are also several special-purpose **troff** commands for local motion. We have already seen \0, which is an unpaddable white space of the same width as a digit. 'Unpaddable' means that it will never be widened or split across a line by line justification and filling. There is also \(blank), which is an unpaddable character the width of a space, \|, which is half that width, \^, which is one quarter of the width of a space, and \&, which has zero width. (This last one is useful, for example, in entering a text line which would otherwise begin with a '.'.)

The command \o, used like

```
\o'set of characters'
```

causes (up to 9) characters to be overstruck, centered on the widest. This is nice for accents, as in

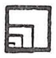
```
syst\o"e\(ga"me t\o"e\(aa"l\o"e\(aa"phonique
```

which makes

système téléphonique

The accents are \(ga and \(aa, or \` and \'; remember that each is just one character to **troff**.

You can make your own overstrikes with another special convention, \z, the zero-motion command. \zx suppresses the normal horizontal motion after printing the single character x, so another character can be laid on top of it. Although sizes can be changed within \o, it centers the characters on the widest, and there can be no horizontal or vertical motions, so \z may be the only way to get what you want:



is produced by

```
.sp 2
\s8\z\(sq\s14\z\(sq\s22\z\(sq\s36\(sq
```

The .sp is needed to leave room for the result.

As another example, an extra-heavy semicolon that looks like

; instead of ; or ;

can be constructed with a big comma and a big period above it:

```
\s+6\z,\v'−0.25m'.\v'0.25m'\s0
```

'0.25m' is an empirical constant.

A more ornate overstrike is given by the bracketing function \b, which piles up characters vertically, centered on the current baseline. Thus we can get big brackets, constructing them with piled-up smaller pieces:

$$\left\{\left[\ x\ \right]\right\}$$

by typing in only this:

```
.sp
\b'\(lt\(lk\(lb' \b'\(lc\(lf' x \b'\(rc\(rf' \b'\(rt\(rk\(rb'
```

troff also provides a convenient facility for drawing horizontal and vertical lines of arbitrary length with arbitrary characters. \l'1i' draws a line one inch long, like this: ________________. The length can be followed by the character to use if the _ isn't appropriate; \l'0.5i.' draws a half-inch line of dots: .... The construction \L is entirely analogous, except that it draws a vertical line instead of horizontal.

7. Strings

Obviously if a paper contains a large number of occurrences of an acute accent over a letter 'e', typing \o"e\'" for each é would be a great nuisance.

Fortunately, **troff** provides a way in which you can store an arbitrary collection of text in a 'string', and thereafter use the string name as a shorthand for its contents. Strings are one of several **troff** mechanisms whose judicious use lets you type a document with less effort and organize it so that extensive format changes can be made with few editing changes.

A reference to a string is replaced by whatever text the string was defined as. Strings are defined with the command .ds. The line

```
.ds e \o"e\'"
```

defines the string e to have the value \o"e\'"

String names may be either one or two characters long, and are referred to by *x for one character names or *(xy for two character names. Thus to get téléphone, given the definition of the string e as above, we can say t*el*ephone.

If a string must begin with blanks, define it as

```
.ds xx "     text
```

The double quote signals the beginning of the definition. There is no trailing quote; the end of the line terminates the string.

A string may actually be several lines long; if **troff** encounters a \ at the end of *any* line, it is thrown away and the next line added to the current one. So you can make a long string simply by ending each line but the last with a backslash:

```
.ds xx this \
is a very \
long string
```

Strings may be defined in terms of other strings, or even in terms of themselves; we will discuss some of these possibilities later.

8. Introduction to Macros

Before we can go much further in **troff**, we need to learn a bit about the macro facility. In its simplest form, a macro is just a shorthand notation quite similar to a string. Suppose we want every paragraph to start in exactly the same way — with a space and a temporary indent of two ems:

```
.sp
.ti +2m
```

Then to save typing, we would like to collapse these into one shorthand line, a **troff** 'command' like

```
.PP
```

that would be treated by **troff** exactly as

```
.sp
.ti +2m
```

.PP is called a *macro*. The way we tell **troff** what .PP means is to *define* it with the .de command:

```
.de PP
.sp
.ti +2m
..
```

The first line names the macro (we used '.PP' for 'paragraph', and upper case so it wouldn't conflict with any name that **troff** might already know about). The last line .. marks the end of the definition. In between is the text, which is simply inserted whenever **troff** sees the 'command' or macro call

```
.PP
```

A macro can contain any mixture of text and formatting commands.

The definition of .PP has to precede its first use; undefined macros are simply ignored. Names are restricted to one or two characters.

Using macros for commonly occurring sequences of commands is critically important. Not only does it save typing, but it makes later changes much easier. Suppose we decide that the paragraph indent is too small, the vertical space is much too big, and roman font should be forced. Instead of changing the whole document, we need only change the definition of .PP to something like

```
.de PP          \" paragraph macro
.sp 2p
.ti +3m
.ft R
..
```

and the change takes effect everywhere we used .PP.

\" is a **troff** command that causes the rest of the line to be ignored. We use it here to add comments to the macro definition (a wise idea once definitions get complicated).

As another example of macros, consider these two which start and end a block of offset, unfilled text, like most of the examples in this paper:

```
.de BS      \" start indented block
.sp
.nf
.in +0.3i
..
.de BE      \" end indented block
.sp
.fi
.in −0.3i
..
```

Now we can surround text like

```
Copy to
John Doe
Richard Roberts
Stanley Smith
```

by the commands .BS and .BE, and it will come out as it did above. Notice that we indented by .in +0.3i instead of .in 0.3i. This way we can nest our uses of .BS and BE to get blocks within blocks.

If later on we decide that the indent should be 0.5i, then it is only necessary to change the definitions of .BS and .BE, not the whole paper.

9. Titles, Pages and Numbering

This is an area where things get tougher, because nothing is done for you automatically. Of necessity, some of this section is a cookbook, to be copied literally until you get some experience.

Suppose you want a title at the top of each page, saying just

```
‾‾‾‾left top          center top          right top‾‾‾‾
```

In **roff**, one can say

```
.he ′left top′center top′right top′
.fo ′left bottom′center bottom′right bottom′
```

to get headers and footers automatically on every page. Alas, this doesn't work in **troff**, a serious hardship for the novice. Instead you have to do a lot of specification.

You have to say what the actual title is (easy); when to print it (easy enough); and what to do at and around the title line (harder). Taking these in reverse order, first we define a macro .NP (for 'new page') to process titles and the like at the end of one page and the beginning of the next:

```
.de NP
′bp
′sp 0.5i
.tl ′left top′center top′right top′
′sp 0.3i
..
```

To make sure we're at the top of a page, we issue a 'begin page' command ′bp, which causes a skip to top-of-page (we'll explain the ′ shortly). Then we space down half an inch, print the title (the use of .tl should be self explanatory; later we will discuss parameterizing the titles), space another 0.3 inches, and we're done.

To ask for .NP at the bottom of each page, we have to say something like 'when the text is within an inch of the bottom of the page, start the processing for a new page.' This is done with a 'when' command .wh:

```
.wh  −1i  NP
```

(No '.' is used before NP; this is simply the name of a macro, not a macro call.) The minus sign means 'measure up from the bottom of the page', so '−1i' means 'one inch from the bottom'.

The .wh command appears in the input outside the definition of .NP; typically the input would be

```
.de NP
...
..
.wh −1i NP
```

Now what happens? As text is actually being output, **troff** keeps track of its vertical position on the page, and after a line is printed within one inch from the bottom, the .NP macro is activated. (In the jargon, the .wh command sets a *trap* at the specified place, which is 'sprung' when that point is passed.) .NP causes a skip to the top of the next page (that's what the ′bp was for), then prints the title with the appropriate margins.

Why ′bp and ′sp instead of .bp and .sp? The answer is that .sp and .bp, like several other commands, cause a *break* to take place. That is, all the input text collected but not yet printed is flushed out as soon as possible, and the next input line is guaranteed to start a new line of output. If we had used .sp or .bp in the .NP macro, this would cause a break in the middle of the current output line when a new page is started. The effect would be to print the left-over part of that line at the top of the page, followed by the next input line on a new output line. This is *not* what we want. Using ′ instead of . for a command tells **troff** that no break is to take place — the output line currently being filled should *not* be forced out before the space or new page.

The list of commands that cause a break is short and natural:

```
.bp   .br   .ce   .fi   .nf   .sp   .in   .ti
```

All others cause *no* break, regardless of whether you use a . or a '. If you really need a break, add a .br command at the appropriate place.

One other thing to beware of — if you're changing fonts or point sizes a lot, you may find that if you cross a page boundary in an unexpected font or size, your titles come out in that size and font instead of what you intended. Furthermore, the length of a title is independent of the current line length, so titles will come out at the default length of 6.5 inches unless you change it, which is done with the .lt command.

There are several ways to fix the problems of point sizes and fonts in titles. For the simplest applications, we can change .NP to set the proper size and font for the title, then restore the previous values, like this:

```
.de NP
'bp
'sp 0.5i
.ft R           \" set title font to roman
.ps 10          \" and size to 10 point
.lt 6i          \" and length to 6 inches
.tl 'left'center'right'
.ps             \" revert to previous size
.ft P           \" and to previous font
'sp 0.3i
..
```

This version of .NP does *not* work if the fields in the .tl command contain size or font changes. To cope with that requires **troff**'s 'environment' mechanism, which we will discuss in Section 13.

To get a footer at the bottom of a page, you can modify .NP so it does some processing before the 'bp command, or split the job into a footer macro invoked at the bottom margin and a header macro invoked at the top of the page. These variations are left as exercises.

Output page numbers are computed automatically as each page is produced (starting at 1), but no numbers are printed unless you ask for them explicitly. To get page numbers printed, include the character % in the .tl line at the position where you want the number to appear. For example

```
.tl ''- % -''
```

centers the page number inside hyphens, as on this page. You can set the page number at any time with either .bp n, which immediately starts a new page numbered n, or with .pn n, which sets the page number for the next page but doesn't cause a skip to the new page. Again, .bp +n sets the page number to n more than its current value; .bp means .bp +1.

10. Number Registers and Arithmetic

troff has a facility for doing arithmetic, and for defining and using variables with numeric values, called *number registers*. Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. And of course they serve for any sort of arithmetic computation.

Like strings, number registers have one or two character names. They are set by the **.nr** command, and are referenced anywhere by **\nx** (one character name) or **\n(xy** (two character name).

There are quite a few pre-defined number registers maintained by **troff**, among them **%** for the current page number; **nl** for the current vertical position on the page; **dy**, **mo** and **yr** for the current day, month and year; and **.s** and **.f** for the current size and font. (The font is a number from 1 to 4.) Any of these can be used in computations like any other register, but some, like **.s** and **.f**, cannot be changed with **.nr**.

As an example of the use of number registers, in the −ms macro package [4], most significant parameters are defined in terms of the values of a handful of number registers. These include the point size for text, the vertical spacing, and the line and title lengths. To set the point size and vertical spacing for the following paragraphs, for example, a user may say

```
.nr PS 9
.nr VS 11
```

The paragraph macro .PP is defined (roughly) as follows:

```
.de PP
.ps \\n(PS      \" reset size
.vs \\n(VSp     \" spacing
.ft R           \" font
.sp 0.5v        \" half a line
.ti +3m
..
```

This sets the font to Roman and the point size and line spacing to whatever values are stored in the number registers PS and VS.

Why are there two backslashes? This is the eternal problem of how to quote a quote. When **troff** originally reads the macro definition, it peels off one backslash to see what's coming next. To ensure that another is left in the definition when the macro is *used*, we have to put in two backslashes in the definition. If only one backslash is used, point size and vertical spacing will be frozen at the time the macro is defined, not when it is used.

Protecting by an extra layer of backslashes is only needed for \n, *, \$ (which we haven't come to yet), and \ itself. Things like \s, \f, \h, \v, and so on do not need an extra backslash, since they are converted by **troff** to an internal code immediately upon being seen.

Arithmetic expressions can appear anywhere that a number is expected. As a trivial example,

```
.nr PS \\n(PS−2
```

decrements PS by 2. Expressions can use the arithmetic operators +, −, *, /, % (mod), the relational operators >, >=, <, <=, =, and != (not equal), and parentheses.

Although the arithmetic we have done so far has been straightforward, more complicated things are somewhat tricky. First, number registers hold only integers. **troff** arithmetic uses truncating integer division, just like Fortran. Second, in the absence of parentheses, evaluation is done left-to-right without any operator precedence (including relational operators). Thus

```
7*−4+3/13
```

becomes '−1'. Number registers can occur anywhere in an expression, and so can scale indicators like p, i, m, and so on (but no spaces). Although integer division causes truncation, each number and its scale indicator is converted to machine units (1/432 inch) before any arithmetic is done, so 1i/2u evaluates to 0.5i correctly.

The scale indicator u often has to appear when you wouldn't expect it — in particular, when arithmetic is being done in a context that implies horizontal or vertical dimensions. For example,

```
.ll 7/2i
```

would seem obvious enough — 3½ inches. Sorry. Remember that the default units for horizontal parameters like .ll are ems. That's really '7 ems / 2 inches', and when translated into machine units, it becomes zero. How about

```
.ll 7i/2
```

Sorry, still no good — the '2' is '2 ems', so '7i/2' is small, although not zero. You *must* use

```
.ll 7i/2u
```

So again, a safe rule is to attach a scale indicator to every number, even constants.

For arithmetic done within a .nr command, there is no implication of horizontal or vertical dimension, so the default units are 'units', and 7i/2 and 7i/2u mean the same thing. Thus

```
.nr ll 7i/2
.ll \\n(llu
```

does just what you want, so long as you don't forget the u on the .ll command.

11. Macros with arguments

The next step is to define macros that can change from one use to the next according to parameters supplied as arguments. To make this work, we need two things: first, when we define the macro, we have to indicate that some parts of it will be provided as arguments when the macro is called. Then when the macro is called we have to provide actual arguments to be plugged into the definition.

Let us illustrate by defining a macro .SM that will print its argument two points smaller than the surrounding text. That is, the macro call

```
.SM TROFF
```

will produce TROFF.

The definition of .SM is

```
.de SM
\s−2\\$1\s+2
..
```

Within a macro definition, the symbol \\$n refers to the nth argument that the macro was called with. Thus \\$1 is the string to be placed in a smaller point size when .SM is called.

As a slightly more complicated version, the following definition of .SM permits optional second and third arguments that will be printed in the normal size:

```
.de SM
\\$3\s−2\\$1\s+2\\$2
..
```

Arguments not provided when the macro is called are treated as empty, so

```
.SM TROFF ),
```

produces TROFF), while

```
.SM TROFF ). (
```

produces (TROFF). It is convenient to reverse the order of arguments because trailing punctuation is much more common than leading.

By the way, the number of arguments that a macro was called with is available in number register .$.

The following macro .BD is the one used to make the 'bold roman' we have been using for **troff** command names in text. It combines horizontal motions, width computations, and argument rearrangement.

```
.de BD
\&\\$3\f1\\$1\h'-\w'\\$1'u+1u'\\$1\fP\\$2
..
```

The \h and \w commands need no extra backslash, as we discussed above. The \& is there in case the argument begins with a period.

Two backslashes are needed with the \\$n commands, though, to protect one of them when the macro is being defined. Perhaps a second example will make this clearer. Consider a macro called .SH which produces section headings rather like those in this paper, with the sections numbered automatically, and the title in bold in a smaller size. The use is

```
.SH "Section title ..."
```

(If the argument to a macro is to contain blanks, then it must be *surrounded* by double quotes, unlike a string, where only one leading quote is permitted.)

Here is the definition of the .SH macro:

```
.nr SH 0       \" initialize section number
.de SH
.sp 0.3i
.ft B
.nr SH \\n(SH+1    \" increment number
.ps \\n(PS-1       \" decrease PS
\\n(SH. \\$1       \" number. title
.ps \\n(PS         \" restore PS
.sp 0.3i
.ft R
..
```

The section number is kept in number register SH, which is incremented each time just before it is used. (A number register may have the same name as a macro without conflict but a string may not.)

We used \\n(SH instead of \n(SH and \\n(PS instead of \n(PS. If we had used \n(SH, we would get the value of the register at the time the macro was *defined,* not at the time it was *used.* If that's what you want, fine, but not here. Similarly, by using \\n(PS, we get the point size at the time the macro is called.

As an example that does not involve numbers, recall our .NP macro which had a

```
.tl 'left'center'right'
```

We could make these into parameters by using instead

```
.tl '\\*(LT'\\*(CT'\\*(RT'
```

so the title comes from three strings called LT, CT and RT. If these are empty, then the title will be a blank line. Normally CT would be set with something like

```
.ds CT - % -
```

to give just the page number between hyphens (as on the top of this page), but a user could supply private definitions for any of the strings.

12. Conditionals

Suppose we want the .SH macro to leave two extra inches of space just before section 1, but nowhere else. The cleanest way to do that is to test inside the .SH macro whether the section number is 1, and add some space if it is. The .if command provides the conditional test that we can add just before the heading line is output:

```
.if \\n(SH=1 .sp 2i      \" first section only
```

The condition after the .if can be any arithmetic or logical expression. If the condition is logically true, or arithmetically greater than zero, the rest of the line is treated as if it were text — here a command. If the condition is false, or zero or negative, the rest of the line is skipped.

It is possible to do more than one command if a condition is true. Suppose several operations are to be done before section 1. One possibility is to define a macro .S1 and invoke it if we are about to do section 1 (as determined by an .if).

```
.de S1
--- processing for section 1 ---
..
.de SH
...
.if \\n(SH=1 .S1
...
..
```

An alternate way is to use the extended form of the .if, like this:

```
.if \\n(SH=1 \{--- processing
for section 1 ----\}
```

The braces \{ and \} must occur in the positions shown or you will get unexpected extra lines in your output. troff also provides an 'if-else' construction, which we will not go into here.

A condition can be negated by preceding it with !; we get the same effect as above (but less clearly) by using

```
.if !\\n(SH>1 .S1
```

There are a handful of other conditions that can be tested with .if. For example, is the current page even or odd?

```
.if e .tl "even page title"
.if o .tl "odd page title"
```

gives facing pages different titles when used inside an appropriate new page macro.

Two other conditions are t and n, which tell you whether the formatter is **troff** or **nroff**.

```
.if t troff stuff ...
.if n nroff stuff ...
```

Finally, string comparisons may be made in an **.if**:

```
.if 'string1'string2' stuff
```

does 'stuff' if *string1* is the same as *string2*. The character separating the strings can be anything reasonable that is not contained in either string. The strings themselves can reference strings with *, arguments with \$, and so on.

13. Environments

As we mentioned, there is a potential problem when going across a page boundary: parameters like size and font for a page title may well be different from those in effect in the text when the page boundary occurs. **troff** provides a very general way to deal with this and similar situations. There are three 'environments', each of which has independently settable versions of many of the parameters associated with processing, including size, font, line and title lengths, fill/nofill mode, tab stops, and even partially collected lines. Thus the titling problem may be readily solved by processing the main text in one environment and titles in a separate one with its own suitable parameters.

The command .ev n shifts to environment n; n must be 0, 1 or 2. The command .ev with no argument returns to the previous environment. Environment names are maintained in a stack, so calls for different environments may be nested and unwound consistently.

Suppose we say that the main text is processed in environment 0, which is where **troff** begins by default. Then we can modify the new page macro .NP to process titles in environment 1 like this:

```
.de NP
.ev 1          \" shift to new environment
.lt 6i         \" set parameters here
.ft R
.ps 10
... any other processing ...
.ev            \" return to previous environment
..
```

It is also possible to initialize the parameters for an environment outside the .NP macro, but the version shown keeps all the processing in one place and is thus easier to understand and change.

14. Diversions

There are numerous occasions in page layout when it is necessary to store some text for a period of time without actually printing it. Footnotes are the most obvious example: the text of the footnote usually appears in the input well before the place on the page where it is to be printed is reached. In fact, the place where it is output normally depends on how big it is, which implies that there must be a way to process the footnote at least enough to decide its size without printing it.

troff provides a mechanism called a diversion for doing this processing. Any part of the output may be diverted into a macro instead of being printed, and then at some convenient time the macro may be put back into the input.

The command .di xy begins a diversion — all subsequent output is collected into the macro xy until the command .di with no arguments is encountered. This terminates the diversion. The processed text is available at any time thereafter, simply by giving the command

```
.xy
```

The vertical size of the last finished diversion is contained in the built-in number register dn.

As a simple example, suppose we want to implement a 'keep-release' operation, so that text between the commands .KS and .KE will not be split across a page boundary (as for a figure or table). Clearly, when a .KS is encountered, we have to begin diverting the output so we can find out how big it is. Then when a .KE is seen, we decide whether the diverted text will fit on the current page, and print it either there if it fits, or at the top of the next page if it doesn't. So:

```
.de KS     \" start keep
.br        \" start fresh line
.ev 1      \" collect in new environment
.fi        \" make it filled text
.di XX     \" collect in XX
..

.de KE     \" end keep
.br        \" get last partial line
.di        \" end diversion
.if \\n(dn>=\\n(.t .bp  \" bp if doesn't fit
.nf        \" bring it back in no-fill
.XX        \" text
.ev        \" return to normal environment
..
```

Recall that number register nl is the current position on the output page. Since output was being diverted, this remains at its value when the diversion started. dn is the amount of text in the diversion; .t (another built-in register) is the distance to the next trap, which we assume is at the bottom margin of the page. If the diversion is large enough to go past the trap, the .if is satisfied, and a .bp is issued. In either case, the diverted output is then brought back with .XX. It is essential to bring it back in no-fill mode so **troff** will do no further processing on it.

This is not the most general keep-release, nor is it robust in the face of all conceivable inputs, but it would require more space than we have here to write it in full generality. This section is not intended to teach everything about diversions, but to sketch out enough that you can read existing macro packages with some comprehension.

Acknowledgements

I am deeply indebted to J. F. Ossanna, the author of **troff**, for his repeated patient explanations of fine points, and for his continuing willingness to adapt **troff** to make other uses easier. I am also grateful to Jim Blinn, Ted Dolotta, Doug McIlroy, Mike Lesk and Joel Sturman for helpful comments on this paper.

References

[1] J. F. Ossanna, *NROFF/TROFF* User's Manual, Bell Laboratories Computing Science Technical Report 54, 1976.

[2] B. W. Kernighan, *A System for Typesetting Mathematics — User's Guide (Second Edition)*, Bell Laboratories Computing Science Technical Report 17, 1977.

[3] M. E. Lesk, *TBL — A Program to Format Tables*, Bell Laboratories Computing Science Technical Report 49, 1976.

[4] M. E. Lesk, *Typing Documents on UNIX*, Bell Laboratories, 1978.

[5] J. R. Mashey and D. W. Smith, *PWB/MM — Programmer's Workbench Memorandum Macros*, Bell Laboratories internal memorandum.

Berkeley Font Catalog

October 1980

Introduction

This catalog gives samples of the various fonts available at Berkeley using vtroff on our Versatec and Varian. We have them working 4 pages across in a 36 inch Versatec, and rotated 90 degrees on a Benson-Varian 11 inch plotter. The same software should be adaptable to an 11 inch Versatec, and in fact is running at several other sites, however, not having one here, it isn't part of this distribution. Such a driver is available from Tom Ferrin at UCSF.

To use these fonts:

(1) Hershey. This is the default font. The Hershey font is currently the *only* complete font, with all 16 point sizes and all the special characters troff knows about. To get it, use vtroff directly. To illustrate this with the −ms macro package:

```
vtroff −ms paper.nr
```

(2) Fonts with roman, italic, and bold, such as nonie. You can load all three fonts with, for example:

```
vtroff −F nonie −ms paper.nr
```

To get just one of these fonts, use (3) below, appending .r, .i, or .b to the font name to specify which font you want mounted, e.g., to get italics in delegate,

```
vtroff −2 delegate.i −ms paper.nr
```

(3) To get a font without a complete set, choose which font (1, 2, or 3) you want replaced by the chosen font. For example, to use bocklin as though it were bold, since font 3 is bold, use:

```
vtroff −3 bocklin −ms paper.nr
```

To switch between fonts in troff, use

```
.ft 3
```

to switch to font 3, for example, or use

```
\f3word\f1
```

to switch within a line. For more information see the Nroff/Troff Users Manual.

Special note: *troff* thinks it is talking to a CAT phototypesetter. Thus, it does all sorts of strange things, such as enforcing restrictions like 7.54 inches maximum width, 4 fonts, a certain 16 point sizes, proportional spacing by point size, etc.

In particular, the following glyphs will *always* be taken from the special font, no matter what font you are using at the time:

@, #, ", ´, `, <, >, \, {, }, ~, ^, and _

This may explain what are otherwise surprising results in some of the subsequent pages.

In addition, the following Greek letters have been decreed by *troff* as looking so much like their Roman counterparts that the Roman version (font 1) is always printed, no matter what font is mounted on font 1 at the time:

A, B, E, Z, H, I, K, M, N, O, P, T, X.

(See table II in the back of the Nroff/Troff Users's Manual for details about what glyphs are in each font and how to generate the special glyphs.)

Font Layout Positions

Code	Normal		Special		Code	Normal	Special	
000					100		@	
001	ﬁ	\(fi	∞	\(if	101	A	A	\(*A
002	ﬂ	\(fl	⊇	\(ip	102	B	B	\(*B
003	ﬀ	\(ff	∝	\(pt	103	C	Γ	\(*G
004	-	\-	☞	\(rh	104	D	Δ	\(*D
005	_	\(ru	∪	\(cu	105	E	E	\(*E
006	—	\(em	‾	\(rn	106	F	Z	\(*Z
007	•	\(bu	◎	\(bs	107	G	H	\(*Y
010	□	\(sq	±	\(+-	110	H	Θ	\(*H
011		\(fI	≤	\(<=	111	I	I	\(*I
012		\(fL	≥	\(>=	112	J	K	\(*K
013	°	\(de	√	\(sr	113	K	Λ	\(*L
014	†	\(dg	ς	\(ts	114	L	M	\(*M
015	′	\(fm	∫	\(is	115	M	N	\(*N
016	©	\(co	/	\(sl	116	N	Ξ	\(*C
017	®	\(rg	│	\(bv	117	O	O	\(*O
020	¢	\(ct	⌊	\(lf	120	P	Π	\(*P
021	¼	\(14	⌋	\(rf	121	Q	P	\(*R
022	½	\(12	⌈	\(lc	122	R	Σ	\(*S
023	¾	\(34	⌉	\(rc	123	S	T	\(*T
024			⎧	\(lt	124	T	Υ	\(*U
025			⎩	\(lb	125	U	Φ	\(*F
026			⎫	\(rt	126	V	X	\(*X
027			⎭	\(rb	127	W	Ψ	\(*Q
030			⎨	\(lk	130	X	Ω	\(*W
031			⎬	\(rk	131	Y	‡	\(dd
032			⊂	\(sb	132	Z	|	\(br
033			⊃	\(sp	133	[	⊆	\(ib
034			∩	\(ca	134		\	\e
035			¬	\(no	135	]	○	\(ci
036			☜	\(lh	136		^	^
037			∈	\(mo	137	_	_	_
040	space				140		`	\`
041	!				141	a	α	\(*a
042			"		142	b	β	\(*b
043			#		143	c	γ	\(*g
044	$				144	d	δ	\(*d
045	%				145	e	ε	\(*e
046	&				146	f	ζ	\(*z
047			′		147	g	η	\(*y
050	(		∇	\(gr	150	h	ϑ	\(*h
051	)				151	i	ι	\(*i
052	*		×	\(mu	152	j	κ	\(*k
053	+		+	\(pl	153	k	λ	\(*l
054	,				154	l	μ	\(*m
055	-		−	\(mi	155	m	ν	\(*n
056	.				156	n	ξ	\(*c
057	/		÷	\(di	157	o	ο	\(*o
060	0		≡	\(==	160	p	π	\(*p
061	1		≅	\(~=	161	q	ρ	\(*r
062	2		~	\(ap	162	r	σ	\(*s
063	3		≠	\(!=	163	s	τ	\(*t
064	4		←	\(<-	164	t	υ	\(*u
065	5		→	\(->	165	u	φ	\(*f
066	6		↑	\(ua	166	v	χ	\(*x
067	7		↓	\(da	167	w	ψ	\(*q
070	8		§	\(sc	170	x	ω	\(*w
071	9		*	\(**	171	y	∂	\(pd
072	:				172	z	ϕ	\(es
073	;				173		{	{
074			<		174	|	|	\(or
075			=		175		}	}
076			>		176		~	~
077	?				177			

APL FONT, 10 POINT ONLY

A∝ B⊥C∩D⌊E∊ F_G∇H∆I⍳J·K' L⎕M|N⊤O○P⋆ Q?RρS⌈T~U↓ V∪W X⊃Y↑Z⊂ 01234 56789

(" # $ = × ≥ ∨ ∧ ≤ ≠ + ¨ { } { } ^ ~ _\ ∃ @ → < + / \ . > , <

! → (% → = & → × ' → ' (→ ∨) → ∧ : → ≤ * → ≠ - → + = → ¨ [→ {] → } | → ∃ ' → →
; → < + → ÷ ? → \

Baskerville font, roman, ibold, italic, 12 point only (Called "basker" on line.)

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _\ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _\ | @ ' ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _\ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Bocklin font, 14 and 28 point only.

14 point

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz
01234 56789

‘ ’ ( ) : - = [ ] ' ; / ? . ,

If time be of all things the most precious, wasting time must be, as Poor
Richard says, the greatest prodigality; since, as he elsewhere tells us,
lost time is never found again; and what we call time enough, always
proves little enough: Let us then up and be doing, and doing to the
purpose; so by diligence shall we do more with less perplexity.

28 point (No punctuation except period.)

ABCDE FGHIJ KLMNO PQRST
UVWXYZ abcde fghij klmno pqrst
uvwxyz 01234 56789 .

If time be of all things the most
precious wasting time must be as
Poor Richard says the greatest
prodigality since as he elsewhere
tells us lost time is never found
again and what we call time enough
always proves little enough Let us
then up and be doing and doing to
the purpose so by diligence shall we
do more with less perplexity.

Bodoni font, roman, **bold**, *italic*, 10 point only.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Chess, 18 point only

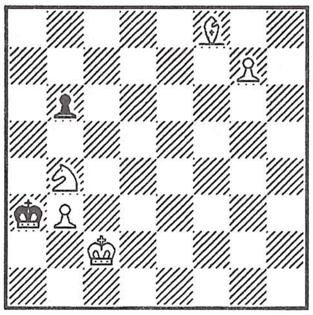
Note: Our attempt at compatibility with Stanford was only 99% successful. If you use a blank space to indicate an empty white square it will come out narrow due to the stupidity of troff. Either include the line

```
        .cs ch 36
```

to put yourself in constant spacing mode or else use zero instead of space. You should also set the vertical spacing to 18 points.

```
.nf
.ft ch
.cs ch 36
.ps 18
.vs 18
HTTTTTTTTX
V0Z0Z0A0ZF
VZ0Z0Z0O0F
V0o0Z0Z0ZF
VZ0Z0Z0Z0F
V0M0Z0Z0ZF
VjPZ0Z0Z0F
V0ZKZ0Z0ZF
VZ0Z0Z0Z0F
WUUUUUUUUG
.sp
.ft P
.ps 8
.cs P
```

p	♟	P	♙
o	▨♟	O	▨♙
b	♝	B	♗
a	▨♝	A	▨♗
n	♞	N	♘
m	▨♞	M	▨♘
r	♜	R	♖
s	▨♜	S	▨♖
q	♛	Q	♕
l	▨♛	L	▨♕
k	♚	K	♔
j	▨♚	J	▨♔
U	—	T	—
F	|	V	|
G	.	W	.
X	.	H	.
0		Z	▨



White mates in three moves.

Clarendon, 14 and 18 point roman only. From SAIL (Paul Martin & Andy Moorer)

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

" # $ Z × ' ( ) : - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

" # $ Z × ' ( ) : - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Computer Modern fonts,roman,*italic*,and bold.(by Don Knuth) 6,7,8,9,10,11,12 point. (Available as cm)

Note that the cm fonts are intended for TEX and don't fare so well with troff. The spacing is not proportional by point size,and hence only one point size can be tuned to be nicely spaced. We have tuned the 10 point size,but the 8 point looks somewhat cramped.

Some of the punctuation is missing in some of the fonts. Knuth also uses a nonstandard notion of ASCII, and hence some glyphs are available only with special symbols such as \(12. Others cannot be accessed at all.

Knuth's fonts somewhat larger than normal, since he intends the output to be reduced before printing. Since troff has a limitation of 7.54 inches width on output, this is not practical. Hence, the original fonts have been relabelled with the point size they are closest to without reduction. Some fonts (6 point bold, 7 point roman, 8 point italic and bold, 9 point bold, and 11 point italic) which would have otherwise been missing were generated by shrinking the next larger point size of the same style. (This goes against the idea of metafont, but we use the tools we have.)

10 Point Roman

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234
56789 ! " # ff% ' ( ) * - { } ^ ~ _ \ @ ⧺ . > , < ' ,, Σ, ,, Ξ, Υ, Φ, Π,,,, Δ, Θ, Λ, Ψ, Ω, ı, ȷ,,,,

If time be of all things the most precious,wasting time must be,as Poor Richard says,the greatest prodigality since,as he elsewhere tells us,lost time is never found again and what we call time enough,always proves little enough Let us then up and be doing,and doing to the purpose so by diligence shall we do more with less perplexity.

10 Point Italic

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234
*56789 ! " # ℘ ∂ ℧ ' ( ) : * - = [ ] { } ^ ~ _ \ ω @ ' ; + / ? . > , < ', ', Σ, -, -, Ξ, Υ, Φ, Π,*
η, θ, ι, Δ, Θ, Λ, Ψ, Ω, α, β, γ, ς ϵ δ

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

10 Point Bold

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234
56789 ! " # ft % & ' ( ) : * - = [ ] { } ^ ~ _ \ fl @ ' ; + / ? . > , < ', ', Σ, -, -, Ξ, Υ, Φ, Π,
˘, ¯, ˉ; Δ, Θ, Λ, Ψ, Ω, ı, ȷ, ˜, ˙, ˙

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

6 Point Roman, **Bold**,and *Italic*.
7 Point Roman, **Bold**, and *Italic*.
8 Point Roman, **Bold**, and *Italic*.
9 Point Roman, **Bold**, and *Italic*.
10 Point Roman, **Bold**, and *Italic*.
11 Point Roman, **Bold**, and *Italic*.
12 Point Roman, **Bold**, and *Italic*.

Countdown (22 point, upper case letters only.) From SAIL (Paul Martin)

ABCDE FGHIJ KLMNO PQRST UVWXYZ

COUNTDOWN HAS NO INTEGERS TO COUNT DOWN WITH BUT IT COMPENSATES BY BEING UGLY AND ILLEGIBLE

Cyrillic, 12 point only

ЖЦъЗ абде фгхи клмно прст увйз

ф тиме бе оф алл тхингс тхе мост прециоус астинг тиме муст бе ас оор ихард сайс тхе греатест продигалитй сике ас хе елсехере теллс ус лост тиме ис невер фоунд агаин анд хат е алл тиме еноугх алайс провес литтле еноугх ет ус тхен уп анд бе доинг анд доинг то тхе пурпосе со бй дилигене скалл е до море итх лесс перплеитй

W→Ж X→Ц Y→ъ Z→З a→а b→б d→д e→е f→ф g→г h→х i→и k→к l→л m→м n→н o→о p→п r→р s→с t→т u→у v→в y→й z→з

Delegate, roman, *italic*, and **bold**, 12 point only

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ↑ ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ / @ ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ‘ ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Fix fixed width font, 6, 9, 10, 12, 14 point

6 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " $ % & ' ( ) : * - = [ ] _ | ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

9 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ‘ ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

10 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ‘ ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

12 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

14 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Gacham, roman, bold, *italic*, 10 point only
The gacham font is almost indistinguishable from the fix font. In fact, it has been pointed out that our gacham roman and bold fonts really *are* fix. Sigh. They are included anyway for convenience.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Greek, 10 point only

This font provides an alternative to the Greek characters on the standard special font.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxy

ΑΒΧΔΕ ΦΓΗΙϕ ΚΛΜΝΟ ΠΘΡΣΤ ΤΩΞΨΖ αβχδε φγηιφ κλμνο πθρστ υςωξψζ

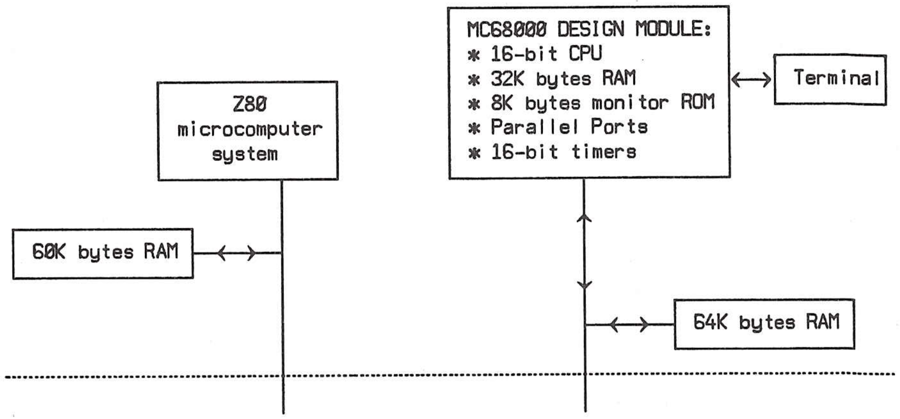
Ιφ τιμε βε οφ αλλ τηινγσ τηε μοστ πρεχιουσ ωαστινγ τιμε μυστ βε ασ Ποορ Ριχηαρδ σαψσ τηε γρεατεστ προδιγαλιτψ σινχε ασ ηε ελσεωηερε τελλσ υσ λοστ τιμε ισ νεςερ φουνδ αγαιν ανδ ωηατ ωε χαλλ τιμε ενουγη αλωαψσ προςεσ λιττλε ενουγη Λετ υσ τηεν υπ ανδ βε δοινγ ανδ δοινγ το τηε πυρποσε σο βψ διλιγενχε σηαλλ ωε δο μορε ωιτη λεσσ περπλεξιτψ

The h19 font includes a subset of the h19's graphic charactor set, plus a few logical extensions to allow forms and diagrams to be drawn. The charactors are the same as the h19's graphic interpratation set.

'	a	b	c	d	e	f	s	t	u	v	m	n	h	i	k	l
|	−	┼	┐	┘	└	┌	┬	┤	┴	├	⋯	┼	→	←	↓	↑

The charactors are designed to overlap.

Example of usage for diagrams:



Hebrew, 16, 24, and 36 point only

16 point

עדסבא עיִהגפ ונמלכ טשרקפ זיחתוו בא יף ןסלפ שפ ייתוו 01234 56789

! " # ף צ . ( ) : ¯ [ ] { } ^ ~ _ \ | @ " ; ץ ? . > , <

עהט שאה טיַ. שטיַטסיַדערפ דנוּפורפ רופ עסיַהס טנעללעסחע נא שיַ טנופ תודאהש עהט
אוֹאודא שאַ : שוֵיִפ ורפ ף : וללו וא שיַ ורף תאֵש. עלבאדאערנו טשומלא גנייַעב פוֹ עגאטנאודא
ף בלי ואלסש אאַבל.

24 point

דסבא הג ע נמלכ טשרקפ זיחתו ם

&→צ .

הט שאהד ת טנ ש נא סח לל טנ הס ם ר רפ דנ
רפ ד טס שנ. ט שאה הט גאטנאודא ב גנ טשמלא
רנ לבאדא .

36 point (rather ragged)

דסבא הג ע נמלכ טשרקפ זיחתו

צ .

10 point Hershey

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789 !, $, %, &, ', (,), :, *, -, [,], ', ;, /, ?, .

\(em → —, – → -, \– → –, \(bu → •, \(sq → ▪, \(ru → _, \(14 → ¼, \(12 → ½, \(34 → ¾, \(fi → fi, \(fl → fl, \(ff → ff, \(Fi → ffi, \(Fl → ffl, \(de → °, \(dg → †, \(fm → ′, \(ct → ¢\(rg → ®
\(co → ©
When you flex your fingers in a coffin, it can baffle a giraffe.

*ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789 !, $, %, &, ', (,), :, *, -, [,], ', ;, /, ?, .*

\(em → —, – → -, \– → –, \(bu → •, \(sq → ▪, \(ru → _, \(14 → ¼\(12 → ½\(34 → ¾\(fi → *fi*, \(fl → *fl*, \(ff → *ff*, \(Fi → *ffi*, \(Fl → *ffl*, \(de → °, \(dg → †, \(fm → ′, \(ct → ¢\(rg → ®\(co → ©
When you flex your fingers in a coffin, it can baffle a giraffe.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789 !, $, %, &, ', (,), :, *, -, [,], ', ;, /, ?, .

\(em → —, – → -, \– → –, \(bu → •, \(sq → ▪, \(ru → _, \(14 → ¼\(12 → ½\(34 → ¾\(fi → **fi**, \(fl → **fl**, \(ff → **ff**, \(Fi → **ffi**, \(Fl → **ffl**, \(de → °, \(dg → **†**, \(fm → ′, \(ct → ¢\(rg → ®\(co → ©
When you flex your fingers in a coffin, it can baffle a giraffe.
From special font: " # = { } ^ ~ _\ | @ ` ´ + > <

Special characters: \(pl → +, \(mi → −, \(eq → =, \(** → ∗, \(sc → §, \(aa → ´, \(ga → `, \(ul → _, \(sl → /, \(*a → α, \(*b → β, \(*g → γ, \(*d → δ, \(*e → ε, \(*z → ζ, \(*y → η, \(*h → ϑ, \(*i → ι, \(*k → κ, \(*l → λ, \(*m → μ, \(*n → ν, \(*c → ξ, \(*o → ο, \(*p → π, \(*r → ρ, \(*s → σ, \(ts → ς, \(*t → τ, \(*u → υ, \(*f → φ, \(*x → χ, \(*q → ψ, \(*w → ω, \(*A → A, \(*B → B, \(*G → Γ, \(*D → Δ, \(*E → E, \(*Z → Z, \(*Y → H, \(*H → Θ, \(*I → I, \(*K → K, \(*L → Λ, \(*M → M, \(*N → N, \(*C → Ξ, \(*O → O, \(*P → Π, \(*R → P, \(*S → Σ, \(*T → T, \(*U → Υ, \(*F → Φ, \(*X → X, \(*Q → Ψ, \(*W → Ω, \(sr → √, \(rn → ‾, \(>= → ≥, \(<= → ≤, \(== → ≡, \(~= → ≃, \(ap → ~, \(!= → ≠, \(-> → →, \(<- → ←, \(ua → ↑, \(da → ↓, \(mu → ×, \(di → ÷, \(+- → ±, \(cu → ∪, \(ca → ∩, \(sb → ⊂, \(sp → ⊃, \(ib → ⊆, \(ip → ⊇, \(if → ∞, \(pd → ∂, \(gr → ∇, \(no → ¬, \(is → ∫, \(pt → ∝, \(eq → =, \(no → ¬, \(br → |, \(dd → ‡, \(rh → ☞\(lh → ☜, \(bs → ● \(or → |, \(ci → ○, \(lt → ⎫, \(lb → ⎩, \(rt → ⎧, \(rb → ⎭, \(lk → ⎨, \(rk → ⎬, \(bv → |, \(lf → ⌊, \(rf → ⌋, \(lc → ⌈, \(rc → ⌉

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

This is an *example* of a sample in **various** fonts.

Hershey font. This is the default font for vtroff. Roman, *Italic* and **Bold** in 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36 point. The following examples are 10 point.

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

6 point Roman, **Bold**, and *Italic*.
7 point Roman, **Bold**, and *Italic*.
8 point Roman, **Bold**, and *Italic*.
9 point Roman, **Bold**, and *Italic*.
10 point Roman, **Bold**, and *Italic*.
11 point Roman, **Bold**, and *Italic*.
12 point Roman, **Bold**, and *Italic*.
14 point Roman, **Bold**, and *Italic*.
16 point Roman, **Bold**, and *Italic*.
18 point Roman, **Bold**, and *Italic*.
20 point Roman, **Bold**, and *Italic*.
22 point Roman, **Bold**, and *Italic*.
24 point Roman, **Bold**, and *Italic*.
28 point Roman, **Bold**, and *Italic*.
36 point Roman, **Bold**, and *Italic*.

Meteor, roman, bold, *italic*, 8, 10, 12 point, no 12 point italic.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ / @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Microgramma font, 10 point only

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Mona font, 24 point only

ABCDE FGHIJ KLMNO PQRST UVWXYZ
abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ ¢ & ' ( ) : - { } ^ ~ _ \ @ ; ? .
> , <

Philadelphia is the most pecksniffian of American cities, and thus probably leads the world.
- H. L. Mencken

Nonie, roman, **bold**, *italic*, 8, 10, 12 point

8 point
ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

10 point
ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

12 point
ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234
56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor
Richard says, the greatest prodigality; since, as he elsewhere tells us, lost
time is never found again; and what we call time enough, always proves little
enough: Let us then up and be doing, and doing to the purpose; so by
diligence shall we do more with less perplexity.

*ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234
56789*

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <*

*If time be of all things the most precious, wasting time must be, as Poor
Richard says, the greatest prodigality; since, as he elsewhere tells us, lost
time is never found again; and what we call time enough, always proves little
enough: Let us then up and be doing, and doing to the purpose; so by
diligence shall we do more with less perplexity.*

**ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz
01234 56789**

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ' ; + / ? . > , <

**If time be of all things the most precious, wasting time must be, as Poor
Richard says, the greatest prodigality; since, as he elsewhere tells us,
lost time is never found again; and what we call time enough, always
proves little enough: Let us then up and be doing, and doing to the
purpose; so by diligence shall we do more with less perplexity.**

Old English, 8, 14, and 18 point only. (This font is called "oldenglish" on line.)

8 point

ABCDE FGHIJ KLMNO PQRST UV WXY abcde fghij klmno pqrst uvwxyz 01234 56789

" #   '   : -   { } ^ ~ _ \ @ ' ;   . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality;since, as he elsewhere tells us, lost time is never found again;and what we call time enough, always proves little enough:Let us then up and be doing, and doing to the purpose;so by diligence shall we do more with less perplexity.

14 point

ABCDE FGHIJ KLMNO PQRST UVWXY abcde fghij klmno pqrst uvwxyz 01234 56789

" #   :   { } ^ ~ _ \ @ ;   . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality;since, as he elsewhere tells us, lost time is never found again;and what we call time enough, always proves little enough:Let us then up and be doing, and doing to the purpose;so by diligence shall we do more with less perplexity.

18 point

ABCDE FGHIJ KLMNO PQRST UVWXYZ
abcde fghij klmno pqrst uvwxyz 01234 56789

" #   ' ( )   -   { } ^ ~ - \   ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality since, as he elsewhere tells us, lost time is nver found again and what we call time enough, always proves little enough and I think I'm wasting time typing all this stuff

PIP FONT, 16 POINT ONLY, NO LOWER CASE

ABCDE FGHIJ KLMNO PQRST UVWXYZ   01234 56789

! " #   ' ( ) : -   { } ^ ~ _ \ @ ' ;   ? . > , <

IT COULD PROBABLY BE SHOWN BY FACTS AND FIGURES THAT THERE IS NO DISTINCTLY NATIVE AMERICAN CRIMINAL CLASS EXCEPT CONGRESS.

-- MARK TWAIN

Playbill font, 10 point only

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 0123

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ @ ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

Script, 18 point only. This font appears to be almost identical to the "Coronet" font from SAIL, except that the period and one other glyph of Coronet are missing a row, and Coronet is supposed to be 16 point. (They are both really the same size.)

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

" #   :   { } ^ ~ _ \ @ ;   . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

SHADOW, 16 POINT ONLY, NO LOWER CASE

ABCDE FGHIJ KLMNO PQRST UVWXYZ   01234 56789

! " #   '   :   [ ] { } ^ ~ _ \ @ ' ; +   . > , <

THE SHADOW FONT IS AN EXCELLENT CHOICE FOR PROFOUND PREDICTIONS. IT HAS THE ADVANTAGE OF BEING ALMOST UNREADABLE.

++ S H A D O W ++

SIGN, 22 POINT ONLY

ABCDE FGHIJ KLMNO PQRST
UVWXYZ   » « 01234 56789

! " #   '   : * - =   { } ^ ~ _ @ ; / . > , <

THIS FONT WAS INVENTED BY A DRAFTSMAN WHO HAD LOST HIS FRENCH CURVE. » SO IT GOES «

LOWER CASE L IS », LOWER CASE R IS «.

Stare hershey font. This font is identical to the hershey font except that the point sizes are one point smaller, and the width tables are those used for the real typesetter. Hence, this font is useful when previewing documents that are to be sent to a typesetter to make sure the spacing, paging, and so on is right. There are Roman, *Italic* and **Bold** in 8, 9, 10, 11, 12, 14, and 16 point. The following examples are 10 point.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

*! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <*

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789

! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <

If time be of all things the most precious, wasting time must be, as Poor Richard says, the greatest prodigality; since, as he elsewhere tells us, lost time is never found again; and what we call time enough, always proves little enough: Let us then up and be doing, and doing to the purpose; so by diligence shall we do more with less perplexity.

8 point Roman, **Bold**, and *Italic*.
9 point Roman, **Bold**, and *Italic*.
10 point Roman, **Bold**, and *Italic*.
11 point Roman, **Bold**, and *Italic*.
12 point Roman, **Bold**, and *Italic*.
14 point Roman, **Bold**, and *Italic*.
16 point Roman, **Bold**, and *Italic*.

Times fonts, roman, *italic*, and **bold**. 10 point only.
These fonts showed up in a directory labelled "timesroman" along with three other fonts which turned out to be nonie, meteor, and news gothic. They are probably not really times fonts, but seem to be pretty close. Notice the top of the "2" for a clear difference from a real Times Roman font.

It is our desire to have a real, digitized version of the times fonts from the phototypesetter. We eventually plan to do this. At that point, the times font will probably replace the hershey font as the default. Such a Times font is already available from Johns Hopkins University for a fee, but we couldn't redistribute it, so we plan do digitize them ourselves.

10 Point

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <
', `, —, -, -, –, ●, □, , ¼, ½, ¾, fi, fl, ff, ffi, ffl, °, †, ', ¢,®©

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <
', `, —, -, -, –, ●, □, , ¼, ½, ¾, fi, fl, ff, ffi, ffl, °, †, ′, ¢,®©

ABCDE FGHIJ KLMNO PQRST UVWXYZ abcde fghij klmno pqrst uvwxyz 01234 56789
! " # $ % & ' ( ) : * - = [ ] { } ^ ~ _ \ | @ ` ; + / ? . > , <
', `, —, -, -, –, ●, ■, , ¼, ½, ¾, fi, fl, ff, ffi, ffl, °, †, ', ¢,®©

A Guide to the Dungeons of Doom

Michael C. Toy
Kenneth C. R. C. Arnold

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

ABSTRACT

Rogue is a visual CRT based fantasy game which runs under the UNIX† timesharing system. This paper describes how to play rogue, and gives a few hints for those who might otherwise get lost in the Dungeons of Doom.

†UNIX is a trademark of Bell Laboratories

1. Introduction

You have just finished your years as a student at the local fighter's guild. After much practice and sweat you have finally completed your training and are ready to embark upon a perilous adventure. As a test of your skills, the local guildmasters have sent you into the Dungeons of Doom. Your task is to return with the Amulet of Yendor. Your reward for the completion of this task will be a full membership in the local guild. In addition, you are allowed to keep all the loot you bring back from the dungeons.

In preparation for your journey, you are given an enchanted mace, a bow, and a quiver of arrows taken from a dragon's hoard in the far off Dark Mountains. You are also outfitted with elf-crafted armor and given enough food to reach the dungeons. You say goodbye to family and friends for what may be the last time and head up the road.

You set out on your way to the dungeons and after several days of uneventful travel, you see the ancient ruins that mark the entrance to the Dungeons of Doom. It is late at night, so you make camp at the entrance and spend the night sleeping under the open skies. In the morning you gather your weapons, put on your armor, eat what is almost your last food, and enter the dungeons.

2. What is going on here?

You have just begun a game of rogue. Your goal is to grab as much treasure as you can, find the Amulet of Yendor, and get out of the Dungeons of Doom alive. On the screen, a map of where you have been and what you have seen on the current dungeon level is kept. As you explore more of the level, it appears on the screen in front of you.

Rogue differs from most computer fantasy games in that it is screen oriented. Commands are all one or two keystrokes[1] and the results of your commands are displayed graphically on the screen rather than being explained in words.[2]

Another major difference between rogue and other computer fantasy games is that once you have solved all the puzzles in a standard fantasy game, it has lost most of its excitement and it ceases to be fun. Rogue, on the other hand, generates a new dungeon every time you play it and even the author finds it an entertaining and exciting game.

3. What do all those things on the screen mean?

In order to understand what is going on in rogue you have to first get some grasp of what rogue is doing with the screen. The rogue screen is intended to replace the "You can see ..." descriptions of standard fantasy games. Figure 1 is a sample of what a rogue screen might look like.

3.1. The bottom line

At the bottom line of the screen are a few pieces of cryptic information describing your current status. Here is an explanation of what these things mean:

Level	This number indicates how deep you have gone in the dungeon. It starts at one and goes up as you go deeper into the dungeon.
Gold	The number of gold pieces you have managed to find and keep with you so far.
Hp	Your current and maximum hit points. Hit points indicate how much damage you can take before you die. The more you get hit in a fight, the lower they get. You can regain hit points by resting. The number in parentheses is the maximum number your hit points can reach.

[1] As opposed to pseudo English sentences.

[2] A minimum screen size of 24 lines by 80 columns is required. If the screen is larger, only the 24x80 section will be used for the map.

```
- - - - - - - - - - - - - -
| . . . . . . . . . . . . +
| . . @ . . . . ] . . .   |
| . . . . . B . . . . . . |
| . . . . . . . . . . . . |
- - - - - - + - - - - - - -


Level: 1  Gold: 0     Hp: 12(12)  Str: 16(16)  Ac: 6  Exp: 1/0
```

Figure 1

Str	Your current strength and maximum ever strength. This can be any integer less than or equal to 31, or greater than or equal to three. The higher the number, the stronger you are. The number in the parentheses is the maximum strength you have attained so far this game.
Ac	Your current armor class. This number indicates how effective your armor is in stopping blows from unfriendly creatures. The lower this number is, the more effective the armor.
Exp	These two numbers give your current experience level and experience points. As you do things, you gain experience points. At certain experience point totals, you gain an experience level. The more experienced you are, the better you are able to fight and to withstand magical attacks.

3.2. The top line

The top line of the screen is reserved for printing messages that describe things that are impossible to represent visually. If you see a "--More--" on the top line, this means that rogue wants to print another message on the screen, but it wants to make certain that you have read the one that is there first. To read the next message, just type a space.

3.3. The rest of the screen

The rest of the screen is the map of the level as you have explored it so far. Each symbol on the screen represents something. Here is a list of what the various symbols mean:

@	This symbol represents you, the adventurer.
-|	These symbols represent the walls of rooms.
+	A door to/from a room.
.	The floor of a room.
#	The floor of a passage between rooms.
*	A pile or pot of gold.
)	A weapon of some sort.
]	A piece of armor.
!	A flask containing a magic potion.
?	A piece of paper, usually a magic scroll.

=	A ring with magic properties
/	A magical staff or wand
^	A trap, watch out for these.
%	A staircase to other levels
:	A piece of food.
A-Z	The uppercase letters represent the various inhabitants of the Dungeons of Doom. Watch out, they can be nasty and vicious.

4. Commands

Commands are given to rogue by typing one or two characters. Most commands can be preceded by a count to repeat them (e.g. typing "10s" will do ten searches). Commands for which counts make no sense have the count ignored. To cancel a count or a prefix, type <ESCAPE>. The list of commands is rather long, but it can be read at any time during the game with the "?" command. Here it is for reference, with a short explanation of each command.

?	The help command. Asks for a character to give help on. If you type a "*", it will list all the commands, otherwise it will explain what the character you typed does.

/	This is the "What is that on the screen?" command. A "/" followed by any character that you see on the level, will tell you what that character is. For instance, typing "/@" will tell you that the "@" symbol represents you, the player.

h, H, ^H
Move left. You move one space to the left. If you use upper case "h", you will continue to move left until you run into something. This works for all movement commands (e.g. "L" means run in direction "l") If you use the "control" "h", you will continue moving in the specified direction until you pass something interesting or run into a wall. You should experiment with this, since it is a very useful command, but very difficult to describe. This also works for all movement commands.

j	Move down.

k	Move up.

l	Move right.

y	Move diagonally up and left.

u	Move diagonally up and right.

b	Move diagonally down and left.

n	Move diagonally down and right.

t	Throw an object. This is a prefix command. When followed with a direction it throws an object in the specified direction. (e.g. type "th" to throw something to the left.)

f	Fight until someone dies. When followed with a direction this will force you to fight the creature in that direction until either you or it bites the big one.

m	Move onto something without picking it up. This will move you one space in the direction you specify and, if there is an object there you can pick up, it won't do it.

z	Zap prefix. Point a staff or wand in a given direction and fire it. Even non-directional staves must be pointed in some direction to be used.

^	Identify trap command. If a trap is on your map and you can't remember what type it is, you can get rogue to remind you by getting next to it and typing "^" followed by the direction that would move you on top of it.

s	Search for traps and secret doors. Examine each space immediately adjacent to you for the existence of a trap or secret door. There is a large chance that even if there is something there, you won't find it, so you might have to search a while before you find something.

>	Climb down a staircase to the next level. Not surprisingly, this can only be done if you are standing on staircase.
<	Climb up a staircase to the level above. This can't be done without the Amulet of Yendor in your possession.
.	Rest. This is the "do nothing" command. This is good for waiting and healing.
*	Inventory. List what you are carrying in your pack.
I	Selective inventory. Tells you what a single item in your pack is.
q	Quaff one of the potions you are carrying.
r	Read one of the scrolls in your pack.
e	Eat food from your pack.
w	Wield a weapon. Take a weapon out of your pack and carry it for use in combat, replacing the one you are currently using (if any).
W	Wear armor. You can only wear one suit of armor at a time. This takes extra time.
T	Take armor off. You can't remove armor that is cursed. This takes extra time.
P	Put on a ring. You can wear only two rings at a time (one on each hand). If you aren't wearing any rings, this command will ask you which hand you want to wear it on, otherwise, it will place it on the unused hand. The program assumes that you wield your sword in your right hand.
R	Remove a ring. If you are only wearing one ring, this command takes it off. If you are wearing two, it will ask you which one you wish to remove,
d	Drop an object. Take something out of your pack and leave it lying on the floor. Only one object can occupy each space. You cannot drop a cursed object at all if you are wielding or wearing it.
c	Call an object something. If you have a type of object in your pack which you wish to remember something about, you can use the call command to give a name to that type of object. This is usually used when you figure out what a potion, scroll, ring, or staff is after you pick it up, or when you want to remember which of those swords in your pack you were wielding.
D	Print out which things you've discovered something about. This command will ask you what type of thing you are interested in. If you type the character for a given type of object (*e.g.* "!" for potion) it will tell you which kinds of that type of object you've discovered (*i.e.*, figured out what they are). This command works for potions, scrolls, rings, and staves and wands.
o	Examine and set options. This command is further explained in the section on options.
^R	Redraws the screen. Useful if spurious messages or transmission errors have messed up the display.
^P	Print last message. Useful when a message disappears before you can read it. This only repeats the last message that was not a mistyped command so that you don't loose anything by accidentally typing the wrong character instead of ^P.
<ESCAPE>	Cancel a command, prefix, or count.
!	Escape to a shell for some commands.
Q	Quit. Leave the game.
S	Save the current game in a file. It will ask you whether you wish to use the default save file. *Caveat*: Rogue won't let you start up a copy of a saved game, and it removes the save file as soon as you start up a restored game. This is to prevent people from saving a

game just before a dangerous position and then restarting it if they die. To restore a saved game, give the file name as an argument to rogue. As in

```
% rogue save_file
```

To restart from the default save file (see below), run

```
% rogue −r
```

v	Prints the program version number.
)	Print the weapon you are currently wielding
]	Print the armor you are currently wearing
=	Print the rings you are currently wearing
@	Reprint the status line on the message line

5. Rooms

Rooms in the dungeons are either lit or dark. If you walk into a lit room, the entire room will be drawn on the screen as soon as you enter. If you walk into a dark room, it will only be displayed as you explore it. Upon leaving a room, all monsters inside the room are erased from the screen. In the darkness you can only see one space in all directions around you. A corridor is always dark.

6. Fighting

If you see a monster and you wish to fight it, just attempt to run into it. Many times a monster you find will mind its own business unless you attack it. It is often the case that discretion is the better part of valor.

7. Objects you can find

When you find something in the dungeon, it is common to want to pick the object up. This is accomplished in rogue by walking over the object (unless you use the "m" prefix, see above). If you are carrying too many things, the program will tell you and it won't pick up the object, otherwise it will add it to your pack and tell you what you just picked up.

Many of the commands that operate on objects must prompt you to find out which object you want to use. If you change your mind and don't want to do that command after all, just type an <ESCAPE> and the command will be aborted.

Some objects, like armor and weapons, are easily differentiated. Others, like scrolls and potions, are given labels which vary according to type. During a game, any two of the same kind of object with the same label are the same type. However, the labels will vary from game to game.

When you use one of these labeled objects, if its effect is obvious, rogue will remember what it is for you. If it's effect isn't extremely obvious you will be asked what you want to scribble on it so you will recognize it later, or you can use the "call" command (see above).

7.1. Weapons

Some weapons, like arrows, come in bunches, but most come one at a time. In order to use a weapon, you must wield it. To fire an arrow out of a bow, you must first wield the bow, then throw the arrow. You can only wield one weapon at a time, but you can't change weapons if the one you are currently wielding is cursed. The commands to use weapons are "w" (wield) and "t" (throw).

7.2. Armor

There are various sorts of armor lying around in the dungeon. Some of it is enchanted, some is cursed, and some is just normal. Different armor types have different armor classes. The lower the armor class, the more protection the armor affords against the blows of monsters. Here is a list of the various armor types and their normal armor class:

Type	*Class*
None	10
Leather armor	8
Studded leather / Ring mail	7
Scale mail	6
Chain mail	5
Banded mail / Splint mail	4
Plate mail	3

If a piece of armor is enchanted, its armor class will be lower than normal. If a suit of armor is cursed, its armor class will be higher, and you will not be able to remove it. However, not all armor with a class that is higher than normal is cursed.

The commands to use weapons are "W" (wear) and "T" (take off).

7.3. Scrolls

Scrolls come with titles in an unknown tongue[3]. After you read a scroll, it disappears from your pack. The command to use a scroll is "r" (read).

7.4. Potions

Potions are labeled by the color of the liquid inside the flask. They disappear after being quaffed. The command to use a scroll is "q" (quaff).

7.5. Staves and Wands

Staves and wands do the same kinds of things. Staves are identified by a type of wood; wands by a type of metal or bone. They are generally things you want to do to something over a long distance, so you must point them at what you wish to affect to use them. Some staves are not affected by the direction they are pointed, though. Staves come with multiple magic charges, the number being random, and when they are used up, the staff is just a piece of wood or metal.

The command to use a wand or staff is "z" (zap)

7.6. Rings

Rings are very useful items, since they are relatively permanent magic, unlike the usually fleeting effects of potions, scrolls, and staves. Of course, the bad rings are also more powerful. Most rings also cause you to use up food more rapidly, the rate varying with the type of ring. Rings are differentiated by their stone settings. The commands to use rings are "P" (put on) and "R" (remove).

7.7. Food

Food is necessary to keep you going. If you go too long without eating you will faint, and eventually die of starvation. The command to use food is "e" (eat).

8. Options

Due to variations in personal tastes and conceptions of the way rogue should do things, there are a set of options you can set that cause rogue to behave in various different ways.

---

[3] Actually, it's a dialect spoken only by the twenty-seven members of a tribe in Outer Mongolia, but you're not supposed to *know* that.

8.1. Setting the options

There are two ways to set the options. The first is with the "o" command of rogue; the second is with the "ROGUEOPTS" environment variable[4].

8.1.1. Using the 'o' command

When you type "o" in rogue, it clears the screen and displays the current settings for all the options. It then places the cursor by the value of the first option and waits for you to type. You can type a <RETURN> which means to go to the next option, a "−" which means to go to the previous option, an <ESCAPE> which means to return to the game, or you can give the option a value. For boolean options this merely involves typing "t" for true or "f" for false. For string options, type the new value followed by a <RETURN>.

8.1.2. Using the ROGUEOPTS variable

The ROGUEOPTS variable is a string containing a comma separated list of initial values for the various options. Boolean variables can be turned on by listing their name or turned off by putting a "no" in front of the name. Thus to set up an environment variable so that **jump** is on, **terse** is off, and the **name** is set to "Blue Meanie", use the command

```
% setenv ROGUEOPTS "jump,noterse,name=Blue Meanie"
```
[5]

8.2. Option list

Here is a list of the options and an explanation of what each one is for. The default value for each is enclosed in square brackets. For character string options, input over fifty characters will be ignored.

terse [*noterse*]
: Useful for those who are tired of the sometimes lengthy messages of rogue. This is a useful option for playing on slow terminals, so this option defaults to *terse* if you are on a slow (1200 baud or under) terminal.

jump [*nojump*]
: If this option is set, running moves will not be displayed until you reach the end of the move. This saves considerable cpu and display time. This option defaults to *jump* if you are using a slow terminal.

flush [*noflush*]
: All typeahead is thrown away after each round of battle. This is useful for those who type far ahead and then watch in dismay as a Bat kills them.

seefloor [*seefloor*]
: Display the floor around you on the screen as you move through dark rooms. Due to the amount of characters generated, this option defaults to *noseefloor* if you are using a slow terminal.

passgo [*nopassgo*]
: Follow turnings in passageways. If you run in a passage and you run into stone or a wall, rogue will see if it can turn to the right or left. If it can only turn one way, it will turn that way. If it can turn either or neither, it will stop. This is followed strictly, which can sometimes lead to slightly confusing occurrences (which is why it defaults to *nopassgo*).

tombstone [*tombstone*]
: Print out the tombstone at the end if you get killed. This is nice but slow, so you can turn it off if you like.

---

[4] On Version 6 systems, there is no equivalent of the ROGUEOPTS feature.

[5] For those of you who use the bourne shell, the commands would be

```
$ ROGUEOPTS="jump,noterse,name=Blue Meanie"
$ export ROGUEOPTS
```

inven [*overwrite*]
: Inventory type. This can have one of three values: *overwrite*, *slow*, or *clear*. With *overwrite* the top lines of the map are overwritten with the list when inventory is requested or when "Which item do you wish to . . .? " questions are answered with a "*". However, if the list is longer than a screenful, the screen is cleared. With *slow*, lists are displayed one item at a time on the top of the screen, and with *clear*, the screen is cleared, the list is displayed, and then the dungeon level is re-displayed. Due to speed considerations, *clear* is the default for terminals without clear-to-end-of-line capabilities.

name [account name]
: This is the name of your character. It is used if you get on the top ten scorer's list.

fruit [*slime-mold*]
: This should hold the name of a fruit that you enjoy eating. It is basically a whimsey that rogue uses in a couple of places.

file [*~/rogue.save*]
: The default file name for saving the game. If your phone is hung up by accident, rogue will automatically save the game in this file. The file name may start with the special character "~" which expands to be your home directory.

9. Scoring

Rogue usually maintains a list of the top scoring people or scores on your machine. Depending on how it is set up, it can post either the top scores or the top players. In the latter case, each account on the machine can post only one non-winning score on this list. If you score higher than someone else on this list, or better your previous score on the list, you will be inserted in the proper place under your current name. How many scores are kept can also be set up by whoever installs it on your machine.

If you quit the game, you get out with all of your gold intact. If, however, you get killed in the Dungeons of Doom, your body is forwarded to your next-of-kin, along with 90% of your gold; ten percent of your gold is kept by the Dungeons' wizard as a fee[6]. This should make you consider whether you want to take one last hit at that monster and possibly live, or quit and thus stop with whatever you have. If you quit, you do get all your gold, but if you swing and live, you might find more.

If you just want to see what the current top players/games list is, you can type

```
% rogue −s
```

10. Acknowledgements

Rogue was originally conceived of by Glenn Wichman and Michael Toy. Ken Arnold and Michael Toy then smoothed out the user interface, and added jillions of new features. We would like to thank Bob Arnold, Michelle Busch, Andy Hatcher, Kipp Hickman, Mark Horton, Daniel Jensen, Bill Joy, Joe Kalash, Steve Maurer, Marty McNary, Jan Miller, and Scott Nelson for their ideas and assistance; and also the teeming multitudes who graciously ignored work, school, and social life to play rogue and send us bugs, complaints, suggestions, and just plain flames. And also Mom.

[6] The Dungeon's wizard is named Wally the Wonder Badger. Invocations should be accompanied by a sizable donative.